Spring 1-1-2016

Securing Secrets and Managing Trust in Modern Computing Applications

Andy Jackson Sayler
University of Colorado at Boulder, andy.sayler@gmail.com

Securing Secrets 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 thesis 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

2016

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

_____
Prof. Dirk Grunwald

_____
Prof. Eric Keller

_____
Prof. John Black

_____
Prof. Sangtae Ha

_____
Prof. Blake Reid

Date _____

The final copy of this thesis 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.

Sayler, Andy (Dissertation)

Securing Secrets and Managing Trust in Modern Computing Applications

Thesis directed by Prof. Dirk Grunwald

The amount of digital data generated and stored by users increases every day. In order to protect this data, modern computing systems employ numerous cryptographic and access control solutions. Almost all of such solutions, however, require the keeping of certain secrets as the basis of their security models. How best to securely store and control access to these secrets is a significant challenge: such secrets must be stored in a manner that protects them from a variety of potentially malicious actors while still enabling the kinds of functionality users expect.

This dissertation discusses a system for isolating secrets from the applications that rely on them and storing these secrets via a standardized, service-oriented secret storage system. This "Secret Storage as a Service" (SSaaS) model allows users to reduce the trust they must place in any single actor while still providing mechanisms to support a range of cloud-based, multi-user, and multi-device use cases.

This dissertation contains the following contributions: an overview of the secret-storage problem and how it relates to the security and privacy of modern computing systems and users, a framework for evaluating the degree by which one must trust various actors across a range of popular use cases and the mechanisms by which this trust can be violated, a description of the SSaaS model and how it helps avoid such trust and security failures, a discussion of how the SSaaS approach can integrate with and improve the security of a range of applications, an overview of Custos – a first-generation SSaaS prototype, an overview of Tutamen – a next-generation SSaaS prototypes, and an exploration of the legal and policy implications of the SSaaS ecosystem.

Dedication

Dedicated to Edward Snowden, Chelsea Manning, Bill Binney, Thomas Drake, Perry Fellwock, Russ Tice, Mark Klein, Thomas Tamm, and all those who have risked and who will risk their lives and livelihoods to expose the privacy and security abuses of governments around the world.

Acknowledgements

This document could not have been completed without the assistance of a variety of individuals. First, thanks to all of my committee members for their oversight, suggestions, and time. Dirk Grunwald, my advisor, provided numerous suggestions and coauthored the Custos and Tutamen papers that predated this document. Eric Keller provided suggestions and assistance on a variety of topics including the original Custos implementation that was undertaken as project for his course. Blake Reid provided the impetus for me to get more involved in the policy side of technology and was kind enough to have me as a member of the Colorado Technology Law and Policy Clinic (TLPC) during the Spring of 2015. John Black and Sangtae Ha have both provided suggestions and answers a variety of technical questions.

Beyond those that served on my committee, many other individuals also provided technical support and insights. Matt Monaco provided the implementation of the Tutamen `dm-crypt` FDE client and is a coauthor on the Tutamen paper. Taylor Andrews provided the Tutamen-backed encrypted Dropbox client implementation and is also a coauthor on the Tutamen paper. Joseph Lorenzo Hall and Erik Stallman both provided mentorship and guidance on a variety of security and policy topics during my time in Washington, DC at the Center for Democracy and Technology.

Finally, a big thanks to my partner Denali Hussin for her continual support throughout my time as a PhD student, as well as her editing support on a range of (overly) verbose technical documents. Thanks also to my parents Mike and Lori for their 27+ years of support and assistance. And thanks to all those who assisted and supported my efforts, but whom I have unintentionally failed to mention here.

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 stored 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 computing 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 surveillance, 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 [231]. Furthermore, the myriad of recently publicized data leaks at large companies (e.g. [18]) as well as ongoing government intrusions into third party user data stores (e.g. [125]) 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. This document aims to refute that idea. Instead, the methods proposed herein aim to 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, this document proposes 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) with storing and regulating access to their private secrets (personal information, encryption keys, etc) on their behalf. Such secrets can then be used to bootstrap the security of other data stores via existing cryptographic techniques. Existing technologies and services can than interface with these SSPs via a standard interface to access user secrets, and by proxy, the data they protect 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 granting feature-rich third party services 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 secrets (and by proxy, to the data such secrets might protect).

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 sacrificing 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, syncing it across multiple personal computing devices, or automating access to such data from a variety of services. 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 [65]. 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 allows Dropbox to download these files to additional user devices or to share them with others 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 [66], 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. Various governments could also decrypt user files stored via

Dropbox should a court compel 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 than not using encryption at all. And the added security 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 such "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 misused, it also has some significant downsides:

(1) It breaks Dropbox's sharing use case. While users could still share encrypted versions of their files with other users via Dropbox, they would also have to exchange the associated encryption key out-of-band in order for their colleague to decrypt and read or update any shared file. This 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 could bootstrap a new Dropbox client device simply by signing into their Dropbox account, users must now both sign into their Dropbox account and manually transfer a copy of their encryption key to the new device to enable the required encryption and decryption operations. This adds an additional step to the Dropbox setup process, potentially driving away users.

(3) If the user ever loses 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 benefits 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. Such manual management requirements pose a non-trivial burden for most users.

Thus, the user must chose between the convenient, but potentially privacy-sacrificing, nature of using Dropbox as it exists today, or the secure, but feature-breaking, nature of using Dropbox with ad-hoc client-side encryption. Neither of these are ideal solutions. An ideal solution must allow the user to leverage the existing convenience and benefits of using Dropbox while also reducing the amount of trust they must place in, or privacy they must sacrifice to, Dropbox itself.

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 verse privacy and control. For example:

Mobile Computing Devices

Phones, tablets, and laptops have become ubiquitous modes of modern computing, storing large amounts of our personal data and carrying out many everyday computations on our behalf. 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 [10] or Google's Compute Engine [111] 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.

Datacenter Infrastructure

Modern data centers are highly automated. It is thus not acceptable to assume a trusted user will be available to enter boot-time passwords in order to bootstrap full disk-encryption

systems. Thus, using disk encryption systems on highly automated servers is difficult, if not impossible, to do securely.

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. This document aims to quantify such deficiencies and suggest potential resolutions via the Secret Storage as a Service model.

1.3 Goals

The goals of the work discussed in this document are threefold:

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

The remainder of this document 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 - Applications: Presents a number of potential SSaaS-backed applications and prototypes.

Chapter 8 - Custos: Presents Custos – a first generation SSaaS prototype, as well as its design and implementation.

Chapter 9 - Tutamen: Presents Tutamen – a next generation SSaaS prototype, as well as its design and implementation.

Chapter 10 - Policy: Discusses the legal and policy implications of the SSaaS model.

Chapter 11 - Conclusion: Overview of contributions and closing thoughts.

Chapter 2

Background

The work presented in this document builds on a number of established topics related to computing, privacy, and security. This chapter touches on the fundamentals of such work. Chapter 4 touches on more directly related work.

2.1 Cryptography

Many of the topics discussed in this dissertation leverage cryptographic primitives as the basis of various security and privacy guarantees. Cryptography represents a security primitive that does not rely on trusting specific people, platforms, or systems in order to securely function. Instead, it requires 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 cryptography and asymmetric cryptography. Symmetric cryptography algorithms function on the principle that a single "key" is used for both halves of each 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 (or to insurmountably large

amounts of computing power). 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 symmetric encryption algorithms include block ciphers such as AES [202] and TwoFish [272] as well as stream ciphers such as (the now deprecated) RC4 [208]. Many block ciphers can also be used as stream ciphers by leveraging operation modes such as CTR and OFB [81].

In addition to encryption and decryption, symmetric cryptography algorithms can also be used to provide secure message integrity and authenticity verification. Such algorithms leverage a secret authentication key to generate a message authentication code (MAC) across a given piece of data. This MAC can then be sent to another user along with the data over which it was generated. Any 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 using mechanisms such as HMAC [157] (e.g. HMAC-SHA-256 [145]) or from existing symmetric block ciphers using mechanisms such as CMAC [33, 71] (e.g. AES-CMAC [287]). In most situations, both encryption and authentication are used in tandem to form a secure channel capable of communicating data that is indecipherable to outsiders and over which any 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 in 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 symmetric encryption and authentication keys 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 several novel cryptographic constructs, including a system for multiple parties to negotiate a symmetric encryption key across an insecure channel [62].[1] These concepts paved the way for the development of asymmetric cryptography (i.e. public-key cryptography): 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 these keys 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. These functions can be quickly solved in one direction, but are computationally difficult to reverse without knowledge of a secret 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 in-

[1] Such methods were actually independently developed by the GCHQ (the British signal-intelligence counterpart to the U.S. National Security Agency) in the early 1970s prior to the publication of Diffie and Hellman's research, but this work was classified and remained a secret until the late 1990s [281].

formation (e.g. one of the factors) is not known. Other systems rely on the computation of discrete logarithms across elliptic curves (ECC cryptography) [151, 193], computations across lattices [6],[2] and beyond.

While Diffie and Hellman proposed a theoretical implementation of a public key cryptography system, the first practical public key cryptography system came a few years later with the publication of the RSA [244] algorithm. Diffie and Hellman's joint key negotiation scheme, however, remains a critical component of asymmetric cryptography systems today – especially those that employ "forward secrecy".[3] Both asymmetric cryptography schemes such as RSA and secure negotiation schemes such as Diffie-Hellman can be used to bootstrap secure communications across an insecure channel by allowing two parties to derive or exchange a mutual secret (e.g. session key) that can then be used to facilitate further secure communication using a symmetric encryption and authentication algorithm. Symmetric algorithms tend to be more performant (and for a given key length, more secure) than asymmetric algorithms, making such split-type cryptography systems desirable.

Asymmetric encryption can be used to build the two additional core 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. Any alteration would result

[2] Lattice-based encryption schemes are believed to be resistant to (currently theoretical) quantum-based attacks that are successful at breaking more traditional factoring and ECC-based schemes.

[3] 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 encryption keys. In short, it guarantees that prior messages can not be decrypted should an adversary manage to compromise a long-term user key at some point in the future. Such systems are commonly implemented by decoupling short-term session keys from long-term authentication keys, using systems such as Diffie-Hellman key derivation to generate the former while relying on the latter for the prevention of Man-in-the-Middle attacks during the initial setup process.

in a verification failure when a member of the public decrypts the message signature - making such alterations detectable by the public at large.

Just as asymmetric encryption gives rise to secure signature algorithms, secure signature algorithms give rise to 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 intended 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 from that user, as the basis of user authentication.

Such systems can be further layered and generalized to build cryptographically secure hierarchical assertion systems attesting to the veracity of a range of data. Such public-key infrastructure (PKI) systems allow the public to extrapolate their trust of a core set of actors (i.e. Certificate Authorities, or CAs) to the any party for which such actors are willing to certify specific facts. For example, a server can trust a specific CA to manually verify the identities of various users (e.g. by having them visit their office and present a photo ID). This CA can then cryptographically sign a combination of their declaration attesting to the identity of each user and a copy of the user's public key. This signed combination of metadata and public key forms a user "certificate" that can be presented to various parties. These parties can then use their knowledge of the trusted CA's public key to verify the identity attestation attached to the user certificate – allowing them trust that the user is who they claim to be. This user can then use their ability to prove they (and only they) control a private key corresponding to their certificate to authenticate to a service. The x509 PKI standard is an example of such a PKI system [52]. The x509 system is widely used to prove the validity of domain name ownership on the Internet via `HTTPS`).

2.1.3 Secret Sharing

Beyond the creation of public key cryptography, one of the other major cryptographic breakthroughs of the last fifty years was the invention of unconditionally secure secret sharing schemes. In particular, Adi Shamir (the 'S' from "RSA") proposed a practical and robust secret sharing scheme in 1979 [274]. Secret sharing is a method for splitting a block of information up into two or more pieces such that holders of any subset of the pieces can not infer any information about the original block as a whole. Shamir Secret Sharing allows a user to divide a block 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. Such "n choose k" threshold systems provide a useful method for distributing information amongst multiple parties in situations where no single party can be fully trusted. These systems can also be used to provide redundancy by selecting n to be greater than k.[4]

Shamir Secret Sharing, unlike other cryptography techniques, does not rely on computational complexity as the basis of its security. Instead, it is unconditionally 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.[5]

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, at least k users must combine their pieces in order to re-compute d.

Shamir Secret Sharing (and related "n choose k" systems) are useful in a wide range of situations where one needs to distribute trust across multiple entities. In particular, secret sharing

[4] Not unlike RAID systems can be used to provide redundancy across multiple disks [227].

[5] I.e., computationally secure systems are potentially breakable if an adversary is afforded enough computing power or the ability to solve certain classes of computationally difficult problems quickly (e.g. via new technology such a quantum computers). Unconditionally secure systems remain unbreakable even to an adversary with unlimited computational power or newfangled computing capabilities.

techniques are leveraged in some cryptographically-based access control systems like that described in [121].

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.[6] 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 defacto key management standard for most cryptographic systems, tends to be extremely challenging for the average user to execute properly. These usability challenges often lead to security failures that have little to do with the quality of the cryptography itself.

The poster child of the "cryptographic system are largely unusable" school of thought is a system for encrypting and cryptographically signing email messages known as PGP [44]. PGP (and its open-source implementation - GnuPG/GPG [153]) is one of the longest running cryptographic security projects. It is also known to have major usability issues, making it largely incomprehensible to all but the most highly trained users [345]. 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 [122]. This has led multiple parties to call for the retirement of PGP [123] and/or to suggest alternatives [38, 73, 215, 115]. It remains to be seen if any of the proposed alternatives will be able to provide (or improve upon) 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 [302]. In all cases, properly obtaining, storing, and controlling access to cryptographic keys and related cryptographic secrets

[6] Usability issues are not limited merely to cryptographic systems. Many computing systems struggle with balancing security and usability [93, 134].

tends to be a task for which the typical user is ill-suited. Thus, while cryptographic systems like S/MIME [240] 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.

Even cryptographic systems that largely avoid burdening the user with key-management responsibilities are subject to numerous usability challenges. For example, take systems such as Tor [63] which are designed to allow users to anonymously browse the Internet or otherwise utilize network resources without leaking their identity. Securely using these systems, while important for subverting censorship and authoritarian regimes, entails overcoming numerous usability challenges. Most of these challenges are related to the fact that all secure systems make assumptions about the manner in which they will be used. If these assumptions violated, the security of the system is unlikely to hold. While such assumptions are necessary in order to make implementing secure systems possible, they do not always align well with the manner in which users might use the given system. When assumptions fail to align with user expectations, security failures result. Tor has been subject to several notable security failings related to mismatches between user behavior and usage limits and assumptions. For example, a Tor user who logs into a site on which they have a known user account (e.g. Facebook or Gmail) while connected to the Tor network risks unmasking themselves not through any failings of Tor itself, but through the voluntary self-identification that occurs when they enter their username on such a site [108].

How best to build systems that are both cryptographically secure and easy to use remains an open and pressing question; a question to which this document attempts to provide an answer.

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 now than ever before. The need to store large amounts of data has also entailed the need to control access to such data. 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 document.

Lacking robust encryption, verification, and access control primitives, early storage and file system technologies often simply neglected data security. The rise of multi-user operating systems like Unix mandated the creation of basic file-system access control schemes. The traditional Unix file access control and permissioning scheme grew out of such early multi-user computing systems as part of the virtual file system (VFS) abstraction [138]. The Unix access control 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 (i.e. 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). ACLs 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, have 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 secu-

rity 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 than Unix permissions or Windows NT ACLs. Many systems combine more than one of these approaches to build a fully featured next-generation storage system.

Kher [150] 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) [255], the Andrew File Systems (AFS) [132], and the Common Internet File System (CIFS/SMB) [190]. 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 systems like SFS [185] or OceanStore [162] 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) [34] or eCryptFS [126] provide basic single-user end-to-end file encryption when coupled with traditional network file systems such as NFS. 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 [194]. These challenges arise from the requirement 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 [103], cepheus [92], and Plutus [142].

Beyond traditional local and networked file systems, many users have turned to cloud-backed storage technologies. System like Dropbox [65] or Google Drive [114] offer mechanisms 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 forgo the traditional file storage model all together in favor of various object storage abstractions. Amazon's S3 [11] and Ceph [135] 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. Systems like 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 [346] 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, computing systems have deployed 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. Most 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 [251]. 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 [254, 174, 285]. In such Unix-like access control systems, each user is a member of one or more groups and each file has an owner and a 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's

owner, the user is a member of the file's group, or the user is neither of these things. This model is flexible enough to support many use cases, 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) schemes were originally proposed in systems such as Multics [251] and later popularized by systems such as VMS and Windows NT. 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 permissions 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.

Existing 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 them lies in which actors within a system have the ability to grant/extend permissions to other actors. In MAC systems, all permissions are set by the system administrator and users have no ability to change these permissions themselves or to 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. 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 centralized management models are common and to tighter control over data is desirable. 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 [178]) 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 [256] 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.

Many authentication systems strive to provide mechanisms to allow a user to authenticate to multiple system using a single account. "Single-sign-on" (SSO) systems such as Kerberos [205, 155] aim to unburden users by allowing the use of a single account across multiple machines. Similarly, SSO systems aim to unburden administrators by allowing the centralized management of user accounts. A more recent variant of this idea has been toward the use of dedicated "identify providers" who provide federated authentication to a suite of third party services. This "federated identity management" (FIM) arrangement allows one entity to delegate the authentication of users to a separate entity, often for the purpose of allowing a user to leverage their existing account with one service provider to authenticate to a separate service provider. System such as Shibboleth [167], SAML [211], and OAuth [212] provide frameworks for performing delegation. Other systems leverage these frameworks to provide additional security guarantees [29]. Like more traditional SSO system, FIM systems aim to increase usability and security by reducing the number of accounts a user must maintain, encouraging the use of stronger passwords (or non-password based authentication mechanism) in the process.

2.5 The Cloud

The previous ten years have seen a major shift in the manner in which users and developers obtain various computing resources. Gone are the days where one must purchase their own hardware or operate their own computing systems. Instead, numerous companies are more than happy to sell you any computing 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

Cloud service providers like Amazon, Google, Microsoft, Rackspace, and IBM sell over $150 billion in cloud services annually [85]. The rapid rise of the cloud computing model is supported by a number of desirable qualities the cloud can provide more effectively than 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 capital expenditures (CAPEX, e.g. large arrays of servers) to regular operational expenditures (OPEX, e.g. a monthly subscription fee). This fact gives rise to a number of potential benefits. Whereas 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 than the traditional CAPEX model.

Flexibility

The "pay-for-what-you-need" cloud purchasing model is also far more flexible than the traditional in-house computing model. While 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 facing systems to require constant load throughout the day. Instead, services tend to see peak usage at certain times each day related to the diurnal cycles of their users. Large international cloud service providers can leverage this fact in ways individual hardware operators can not. In particular, such providers can ensure their underlying hardware is uniformly loaded 24/7/365 by spreading the workloads of a diversity of global tenants across their infrastructure. After all, it is always 5:00 PM somewhere. 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 colocate 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 – a PUE[7] of ≈ 1.1 [112]. On the networking front, such colocation

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

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 with better performance and lower costs than in-house data center deployments.

2.5.2 Service Classes

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

IaaS: "Infrastructure as a Service" systems are 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 of which they may build and implement their own services. This class of cloud services represents the most 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 [10] and Google Compute Engine [111] are both examples of IaaS services.

PaaS: One step up the stack are "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 [110] and Heroku [130] are examples of PaaS offerings.

SaaS: "Software as a Service" sits at the top of the cloud service stack. 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 functionality directly to an end user. Examples of SaaS systems include Dropbox [65], Gmail [116], and Facebook [76].

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 (SOA) [340]. Service oriented architectures abstract a set of useful actions into a service that can be consumed by users. SOA 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 is 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 than it is to buy high-end parts with lower failure rates but disproportionately larger costs [19]. 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 data centers.

Virtualization

Virtualization, the ability to simulate one or more "virtual computers" running atop a single physical computer, is not a new concept [106]. 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 has made it simple and cost effective for cloud-providers to offer their services, slicing discrete physical systems between many paying users. Virtualization also allows providers to separate tenants from any single piece of hardware, allowing providers to migrate tenants between physical systems in order to meet up-time and load balancing goals without the user ever being aware of a change.

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 licensing specialty high-cost software, vendor lock-in, or any number of other barriers to deployment mobility. While the cloud provided cheap, commodity (often virtual) hardware, FOSS provides cheap, commodity software to make such hardware 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 [214].

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 discrete technologies, business cases, and user demands 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 ten years have heralded the rapid expansion of a range 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 users trust various third parties with user data? And, as a corollary, are there mechanisms that allow users to control or reduce this degree of trust?

3.1 Modern Use Cases

Such questions, however, can not be answered in a vacuum. Any proposed data security 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. Understanding the predominant modern use cases is thus a prerequisite to proposing security enhancing technologies. Modern use cases can be divided 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. Security and privacy enhancing solutions must be able to accommodate both sets of use cases if they expect to be widely adopted and used.

3.1.1 Consumer Use Cases

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 and other services which act on user data.

3.1.1.1 Multi-Device

It is common 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. Such users expect to be able access their data from any of their devices. Similarly, many 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 plurality and ephemerality of modern computing devices has lead to the rise of a number of solutions that aim to separate a user's data from the devices on which they access it, ensuring 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 user data on a centralized server and provide a manner for users to access this date from each device (i.e. locker services).

Sync services operate on the premise of storing user data locally on each device while automatically syncing changes to such data across multiple devices. They ensure that the user 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 changes to user data on each device and updates data across all devices when a change on one device is detected. 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 sync updates between devices. The act of reading data already present on a device accesses only the local copy, avoiding the

need to consume bandwidth communicating with an external server. This is a desirable quality in situations where bandwidth would either be cost prohibitive or performance restricting to consume on every read. Sync services also tend to be bandwidth efficient when writing data since they can cache a series of updates locally and sync only 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, a 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 mode 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 copies 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 is 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 and sharing of files, 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 selective-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 than 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 from remote devices. Locker services are reminiscent of more traditional networked file systems such as NFS [255] or SMB [190], but are less tightly coupled to a single administrative domain than were such services. Since locker services store a logically centralized copy of each file, the user is presented with a single view of their data regardless of from which device they chose to access it. Such services have several desirable qualities:

No Local Storage: Locker services store all data in online locations, generally atop datacenter-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. The lack of local copies may also 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 than in sync-based solutions. This property can increase the usefulness of such systems in multi-user scenarios where more than 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 a more centralized control point than a 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 accessing their data. Thus, locker services operate best in situations where network bandwidth is reliable and plentiful. 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 services dominate the multi-device data access solution landscape. Examples of popular centralized sync services today include Dropbox [65] ($\approx$ 300 Million Users [283]), Google Drive [114] ($\approx$ 240 Million Users [283]), and Microsoft OneDrive [189] ($\approx$ 250 Million Users [283]). An example of a decentralized sync service is BitTorrent Sync [32] ($\approx$ 2 Million Users [283]).

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 solutions may provide a kind of locker service functionality. There are also cloud services that provide users with online locker-like data access, e.g. systems like OwnCloud [224] implement the WebDAV protocol [104] for remote file access over the Internet. Similarly, distributed systems like Least Authority's Simple Secure Storage Service [168] (S4, built atop Tahoe-LAFS [346]) 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 [191]. 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 [117] allow users to upload music files which they can then access and stream to multiple devices. Similarly, distributed systems like Popcorn Time [325] 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 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 needs 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 allow 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 allowing users to include devices other than 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 multi-device use case, however, adding support for multi-user sharing requires providing access control primitives in addition to basic file transfer or access 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 allow users to place limits on the degree to which they trust other users. Many services provide fairly traditional file-like access control schemes where each user

is granted read and/or write permissions to a specific piece of data. Data owners can use these permission to craft access control policies for the group of user with which they wish to share data.

It is 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 [76] have extensive support for sharing photos, videos, status messages, and other user-generated content. Such systems also provide the data owner with the ability to place limits on how and with whom data is shared (although the effectiveness of such access control settings is often questionable [137]). Similarly, systems like Google Docs [113] or Microsoft Office Online [188] offer users the ability to interactively compose documents. 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. 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 widely adopted by today's users.

3.1.1.3 Hosted Services and Processing

In addition to the multi-device and multi-user scenarios discussed above, many users 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 executed 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 services 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 compose 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 stats like their day-to-day power consumption [206] or record their exercise habits [84] 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 from a multitude of users to return more useful information than 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 ade-

quately protect their data and services hosted atop third party infrastructure significantly expands the total number of computing systems protected.

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 up to 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 systems and 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 developer's 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 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 starts in order to bootstrap the encryption system. Such systems are not easy to operate in a modern cloud-backed environment and thus are of limited usefulness in such scenarios.

Furthermore, the trend toward headless, ephemeral infrastructure capable of rapidly scaling up or down is driving the adoption of configuration management systems such as Chef [222], Salt [250], or Puppet [235]. Such systems are necessary to help developers automate the otherwise monotonous process of configuring cloud compute resources manually. Indeed, the sheer number of cloud servers involved in most larger scale production deployments makes manual management impossible.[1] Thus, new security and privacy enhancing technologies must be able to operate securely even when their configuration must be automated.

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

[1] This fact has given rise to the sentiment that "servers are cattle, not pets" – meaning developers should focus on automatically managing large numbers of systems instead of carefully curating individual systems.

3.2 Threats to Security and Privacy

The current third party provided cloud-computing trends raise a number of security and privacy related questions. To what degree must users trust each service provider to protect their data? How good are service providers at protecting data? What other threats do modern usage models expose?

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 third party 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. This section presents 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 users place in third parties is to not intentionally misuse the data stored with them, e.g. are third parties leveraging user 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 newsfeeds in order to study the effects of one user's emotions on other users [102]. The study was performed on ≈ 700 users without their 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: In 2014, ride-share app Uber [316] 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 [279]. 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" [226]. 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 lead to the outing of a pregnant teenager to her previously unaware father [131]. Clearly such outcomes are not within the realm of what most shoppers 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 an oversight on the part of the third party. Thus, even if users trust that a third party won't intentionally misuse their data, they must still question whether or not third parties are capable of providing adequate protection for user data. Today's users are generally reliant on third parties to protect the data they store and to faithfully enforce any access control or sharing restrictions a user specifies.

Unfortunately, there are many examples of third party data breaches resulting in the unintended release of user data. 2014 alone saw the release of almost 350 million user identities and associated data online, representing the data of over 10% of internet users globally [306]. 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 [17] were subject to a public release of personal photos they had stored with the service. This leak was the result of a targeted attack on the corresponding users' passwords and iCloud accounts [18]. 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.

Office of Personnel Management Breach: In 2015, the U.S. Office of Personnel Management (OPM) announced that their systems had been breached, exposing the personal data of essentially anyone who had held or currently holds a U.S. Government security clearance [94, 338]. This breach, in addition to having high strategic value to foreign attackers, reveled sensitive personnel data of a huge number of U.S. government employees and contractors. This leak was largely due to the use of old and outdated storage and security systems employed by the OPM.

Anthem and Premera Blue Cross Breaches: In early 2015 two major U.S. 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 [160, 161]. While the details of the breaches were not made 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 flaws. Prominent examples of such flaws include Heartbleed [48], a flaw in OpenSSL [217] that allowed attackers to steal private data from many secure servers, and Shellshock [305], a GNU bash [239] 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 purpose of wide-spread surveillance. In particular, the U.S. National Security Agency (NSA) leaks revealed by Mr. Edward Snowden demonstrate the U.S. government's widespread data surveillance programs targeting popular cloud data providers [125]. Outside the Unites States, government-compelled surveillance and user data access are equally, if not more, pervasive [146]. While such actions, at least in the U.S., raise numerous questions of constitutional legality under the 4th Amendment [323] (see Chapter 10), that 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 five years. It is worth considering several of these examples in order to evaluate how to increase the security and privacy guarantees available atop third party services. Notable instances of government surveillance include:

PRISM and MUSCULAR: The NSA PRISM program was/is a Foreign Intelligence Surveillance Court (FISC) [320] approved system for compelling service providers to provide user data to the government [125]. It is believed to be one of the largest programs used by the government to extract user data from various cloud-based services (e.g. Google, Yahoo, Microsoft, etc). Similarly, MUSCULAR was/is a joint NSA and U.K. Government Communication Headquarters (GCHQ) effort to intercept and monitor traffic traversing Google's and Yahoo's inter-datacenter networks [99]. Prior to MUSCULAR's disclosure,

this intra-datacenter traffic was not generally encrypted, and thus was an ideal point for a third party (e.g. the government) 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 popular 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 U.S. third party doctrine [311]). In August 2013 Lavabit shuttered its service in response to a U.S. 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 [170, 171]. 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.

Apple v. FBI: In response to the 2015 San Bernardino shootings, the FBI attempted to compel Apple to help it decrypt one of the shooters' iPhone [15]. The form of encryption Apple uses to protect the iPhone involves a hardware-linked encryption key that can not be easily extracted from the phone, limiting out-of-band cracking opportunities. Furthermore, this key can not be used on the phone without a user-provided passcode. By default, Apple limits the number of guesses a user may make at this passcode and throttles the speed at which a user may guess passcodes. The FBI wished to compel Apple to update the software on the iPhone so that they could try to guess an unlimited number of passcodes at a high rate of speed [37].[2] Apple was disinclined to acquiesce to this request [51]. The case wound up being dropped by the FBI after they were able to leverage an undisclosed security vulnerability to bypass Apple's passcode guessing limits directly [156, 79]. As in the Lavabit case, this case demonstrates the government's interest in compelling companies

[2] Due to Apple's use of signature-verified code, it is not possible for the FBI (or anyone else) to update the Apple's software themselves. Instead, Apple (or the holder of Apple's cryptographic code signing key)must approve and sign any updated code before the iPhone will run it.

to assist them in accessing private user data, even going so far as to potentially require companies to avoid the use of certain forms of encryption or security-enhancing features that would make such assistance difficult or impossible to provide.

The Great Firewall: Moving beyond U.S. government surveillance, China has long been known to employ one of the most sophisticated web monitoring and content control systems in existence [242]. 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 governments to ensure they can access and monitor user data in the cloud. Security and privacy enhancing systems must therefore account for the fact that service 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 both internal and external network traffic.

3.2.4 Physical Security

The 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 on it was rooted on the premise that the device itself could be kept physically secure (i.e. that 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 designed to be mobile – easily carried by the user

and moved about. The combination of these facts significantly increase the likelihood of a computing device storing user data being lost or stolen. As such, it is critical to 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 than ever before, and privacy-enhancing solutions must account for this fact.

As mentioned previously, physical control over cloud-based services is also not generally possible. Developers are thus forced to run many of our modern services atop hardware under another party's 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 inspect data stored on their servers? How much can users trust computations performed on such systems? 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 challenges demonstrate that any security and privacy enhancing systems designed to protect data in either the cloud or atop user devices must contend with the fact that the physical security of the devices on which they operate is not guaranteed. Such systems must be designed with the trustworthiness (or lack there of) of the physical infrastructure 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 researchers and developers 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 need to support modern use cases places further limitations on the manners in which systems may defend against these threats. These limitations disqualify many existing security solutions and underline the need for new approaches.

Traditionally, the solution to many of the threats discussed in § 3.2 involve the use of cryptography. Cryptography is a desirable security primitive since it allows users to protect their data in a manner that is mathematically secure. Cryptography does not rely on a trusted arbiter for the enforcement of its security guarantees. Instead, it is the intrinsic properties of the underlying math itself that give rise to cryptographic security claims. Cryptography can therefore 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 dm-crypt/LUKS [42, 91] 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 (FDE) systems generally require the user to interactively supply a passphrase 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 passphrases 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. A full disk encryption system that stores a copy of the user passphrase locally to avoid the need for the user to supply it is no more secure than a system without encryption at all. Either an adversary has an unlocked box, or they have both a locked box and the key required to unlock it. Neither scenario results in a security benefit for the owner of the box.

Beyond full-disk encryption schemes, various client-side encryption systems also exist with an aim toward minimizing the need to trust the data storage system. In particular, file-level encryption systems like eCryptfs [126] could conceivably be used to encrypt user data locally before uploading it to a cloud service such as Dropbox. Such an action would serve to greatly reduce the degree to which the user must trust third party providers like Dropbox, and does not impinge the same "human-in-the-loop" issues as FDE system due to the differentiated nature of the client-side encryption use case. Unfortunately, such designs have a new set of flaws. Most notably, they do not mesh well

with many of the desired multi-device and multi-user use cases mentioned previously. For example, a user encrypting data on their laptop and storing the ciphertext on Dropbox will be unable to decrypt and access their data from another device such as a phone or tablet. In this scenario, 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 purpose 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 mechanisms other than Dropbox, why can't they just use the same mechanism for their files as well and avoid Dropbox all together? Likewise, if a user wishes to share data with another 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 such an exchange increases the overhead to multi-user sharing, making it unlikely that most users would employ such tactics in the first place.

Due to these and related issues, most traditional encryption systems are not good fits for popular cloud-based use cases. There is a silver lining to this predicament, however. In these situations it is not the cryptography itself that is flawed. This is fortunate! State-of-the-art cryptography is a powerful tool and it would be a major loss to be unable to leverage it in pursuit of privacy and security enhancing systems. 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 cryptographic solution to allow users to deploy secure 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.

Such cryptographic key storage represents just a subset of the larger secret storage problem. How can a user securely store secrets (encryption keys, passwords, personal data, etc) in a manner

that allows them to access them when and where they desire while also ensuring that no unauthorized access to these secrets is allowed?[3] And once securely stored, how can a user use these secrets to bootstrap security and privacy enhancing solution in a manner that supports a range of desirable use cases? These questions and their proposed solutions form the basis of the work presented in this dissertation.

[3] This question is merely a variant of the always/never paradox often encountered when attempting to build secure systems. The problem was 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 [268]?

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. This chapter describes some of the more pertinent of these efforts.

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 analysis of the various links within and surrounding any given security or privacy enhancing system. Such links must be considered when determining the overall guarantees provided by a given system as well as the exposure of the system to various threats.

Flowerday [86] provides an analysis of 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 discipline related to theories of trust and privacy in government, law, and society [45]. Such 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 focus on traditional technical and operational risks and mitigations. Abbas [1] discusses security and threat models for Internet-connected services. Tsipenyuk [313] focuses on common failures in application-level security. Firesmith [83] 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, Cebula [47] takes a comprehensive look at operational cyber-security risks across technical, human, process, and external factors.

While all of these efforts provide 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 Chapter 5 seeks to correct.

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 (and associated implementations). Other efforts use information theory primitives such as 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 or more trusted third parties.

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 underlying a given access control scheme. 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 fails. Likewise, anyone in control of the underlying OS or third party (e.g. an administrator) automatically gains full control over the access control system and the ability to bypass it.[1] This is an acceptable limitation in many situations, especially those based on centrally managed systems with existing physical and administrative safeguards in place. But in distributed systems or cloud environments 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. Goyal [121] and Bethencourt [27] propose several such 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. Goyal [121] allows users to encrypt documents that can only be decrypted by users possessing specific attribute policies encoded in their keys. Bethencourt [27] extends this concept to allow documents to be encrypted with a full access control policy embedded in the encryption itself, allowing only users whose 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 host 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 ten 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. Systems like CryptDB [232] allow users to search and query encrypted data directly, allowing the storage of such databases on untrusted infrastructure. Other systems like CS2 [143] provide 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 [100]. A number of such systems have been proposed – some of which go so far as to propose the possibility of encrypting entire computer programs that will be executed by fully homomorphic processors [40, 39]. Such system are certainly appealing, but thus far, building one with low enough overhead to be practical for general purpose usage 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 sense,

the available homomorphic systems today are limited to only performing certain operations. Furthermore, while such systems allow the processing of data atop untrusted infrastructure, they do not provide a solution for use cases where one wishes to share plain-text data with other users or otherwise leverage an application that inherently requires data to be decrypted. Homomorphic encryption systems also risk leaking certain classes of data as a side effect of the computation they perform. For example, a homomorphic system may be able to sort data without knowing its contents, but in doing so, it might still learn something about the relative ordering of elements. As in other cases, such systems also fail to provide a general solution for the management of the associated cryptographic keys protecting such operations.

4.2.3 Secure Storage

Beyond data processing, minimizing the degree of third party trust required to securely store data atop cloud infrastructure is 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 common classes of 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 computing systems.

A number of traditional encrypted file systems exist with the design goal of avoiding server-side trust [150]. File systems like CryptoCache [136], RFS [64], and Plutus [142] are all designed to allow users to store files on servers without trusting the server itself. Other systems like Keypad [97] and CleanOS [309] 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.

Distributed storage systems like Depot [181], OceanStore [162], and Tahoe [346] 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 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. The SSaaS ideas proposed in this document 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 numerous highly publicized privacy failures by companies ranging 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 user requirements.

4.3.1 Communication Tools

Many of the recent security and privacy enhancing tools have focused on securing inter-user communication. Both the contents of and meta-data associated with such communication has been the focus of many of the recent NSA leaks [271]. The reaction to such reports has spurred both the creation of new tools as well as the expansion of the use of existing secure communication tools.

Email has long been the bedrock of digital communication mediums. The traditional tools for securing email, OpenPGP [216] and its various implementation (e.g. GnuPG [153] and Symantec PGP [304]), are not new, but they have seen renewed levels of interest as effective mass-surveillance countermeasures. Unfortunately these tools remain no more usable today than they were twenty years ago, leading to the multitude of usability issues discussed in Section 2.2. In response, developers have created tools like Mailpile [73] and Mailvelope [182] with the aim of making PGP-based

email security more friendly for the average user. Similarly, Google and Yahoo are both engaging in efforts aimed at making PGP-like encryption and authentication systems available to their web-mail users [115, 349]. Other systems such as STEED aim to adapt traditional PGP principles to make the primitives simpler for the average end user to deal with [154]. Finally, systems such as KeyBase [149] aim to help users identify the public PGP keys of people with whom they wish to communicate.[2]

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 [123]. Nor are they well suited for securing another major form of modern communication: real-time chat. To overcome these deficiencies, companies have released programs such as TextSecure [215] and ChatSecure [20]. Both systems rely on variants of the Off-the-Record (OTR) secure messaging protocol to achieve encrypted, authenticated, and forward-secure real-time communication between two or more parties [223, 38, 105]. These 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 direct replacements for 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 remain elusive.[3]

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 general methods for reducing trust and managing secrets presented in this dissertation.

4.3.2 Password Managers

One of the most common classes of third party secret storage available today is that of password storage by end user password managers. Due to the well-documented failure of user-chosen

[2] In some ways, KeyBase can be seen as the compliantly opposite of the SSaaS model discussed in this document. Whereas KeyBase aims to make it easy for users to discover and verify the public keys or other users, SSaaS aims to make it easy for users to protect their corresponding private keys.

[3] Although, that has not stopped researcher from proposing some creative solutions to work around the forward secrecy problem, such as systems that force email to "self-destruct" after a certain age [98].

passwords [186, 109, 107, 280], experts now recommend that users leverage password management services. Such services rely on a single strong password to protect a collection of unique and random passwords for each website or service a user must access [270, 159, 41].

Systems such as LastPass [164] and 1Password [5] 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, such systems still require the user to trust a single third party with their credentials, and by proxy, access to their online services and accounts. Nor are such services infallible. There have been several noted flaws and breaches of password management systems over the past few years [95, 230, 278].[4]

Password management systems share some of the same goals proposed 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. This work presents more versatile and generalized solutions to the secret storage problem. Such solutions 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 data storage. As mentioned in Section 3.1, most users 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. Storage services like Dropbox [65] or Google Drive [114] are popular, but these solutions provide the associated third parties with

[4] Fortunately the design of such systems has largely mitigated the worst damages from these breaches [70].

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 storage amenities, developers have created systems such as SpiderOak [288], Tresorit [312], and Wuala [163]. Such services aim to provide users with a Dropbox-like alternative that does not require trusting the storage provider. These services shift the encryption of data onto the client, helping to ensure the server only ever holds an encrypted copy of said data. Nonetheless, the confidentially and privacy claims made by such services have been shown to be dubious [347], largely due to the fact the user must still place a large degree of trust in the backing third party in order to facilitate file sharing with other users. This trust allows the third party to impersonate the other users with whom a user wishes to share data. Such impersonation can be used to mount man-in-the-middle attacks that can be used to regain access to the user data the third party claims not to have.

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 discussed in this document could be leveraged to re-implement such systems while further suppressing reliance on a single trusted third party.

4.4 Key and Secret Management Systems

As mentioned in Section 3.3, one of the main hindrances to bootstrapping secure systems 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 dissertation. This work is not the first to recognize key management as one of the critical components to building flexible and secure privacy and security enhancing systems. Similar secret management systems 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 can't accommodate 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 by 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 [142] 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 onto the client, providing some degree of user-facing key management flexibility, but failing to provide a standardized and general key management system. SFS [185] takes this separation a step further by proposing various standardized mechanisms by which a user might manage keys and an external 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 system such as those proposed in this document. Such applications of standardized secret storage systems are discussed further in Chapter 7.

4.4.2 Key Escrow

Key escrow systems represent a common form of key management system. Key escrow systems aim to fit 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 [274] to accomplish their goals.

Early key escrow systems 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) [59]; e.g. the infamous NSA-backed Clipper chip [344]. 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, researchers proposed distributed key escrow systems that make key escrow requests transparent and leverage 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 [36].

While such distributed key management systems 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 fails to explore the benefits and possibilities of the general purpose secret storage systems described in this document. The secret storage systems described herein could, however, be leveraged to fulfill many of the traditional roles of a distributed key-escrow system.

4.4.3 Automated and Cloud-based Secret Management

The most closely related work to that proposed here is the various automation and cloud-oriented key and secret management systems introduced over the past 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 secret storage primitives available to developers leveraging automated configuration management systems and/or third party cloud platforms.

Rackspace's CloudKeep [238, 236] (now OpenStack Barbican [218]) 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. CloudKeep/Barbican integrates with the OpenStack [219] cloud infrastructure management platform to provide secret storage services to OpenStack users. Such a system, however, still requires the user to trust the OpenStack provider with the storage of their secrets, and does not allow for the party providing secret storage to be easily decoupled from the party providing other cloud services.

Similarly, 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. HSMs are then tasked with performing all necessary cryptographic operations requiring such keys on the user's behalf. Standardized interfaces such as PKCS11 [249] 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 shared computer memory space (and potentially leaking them in the process, as occurred in Heartbleed [48]). 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 [9]. 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 of trust in a single third party to faithfully operate virtual HSMs. The SSaaS work discussed in this dissertation could potentially be used to build similar "soft" HSM systems without requiring single third party trust [177].

Beyond infrastructure-tied services, a number of open source secret storage systems have been developed by industry, including HashiCorp's Vault [129], Lyft's Confidant [179], and Square's Keywhiz [290]. Similar commercial systems also exist with the aim of providing developers with turnkey key-management systems [96, 233, 246]. The development of and demand for such systems demonstrate the clear challenges industry faces when it comes to managing secrets. Such systems, however, have a number of flaws. For example, they generally require placing trust in a single third

party, or in the case of self-hosted solutions, a single server.[5] They also lack the cross-platform standardization necessary to incentivize some of market-driven forms of trust-preserving behavior discussed in Chapters 5 and 6. And they tend to be designed for use within a single administrative domain, limiting their usefulness in loosely-coupled global-scale scenarios.

[5] Although this party may now differ from the party providing the underlying cloud compute or data storage servers.

Chapter 5

An Issue of Trust

Trust and privacy are closely related qualities in computing [86]. Can users maintain their digital privacy without having to trust anyone? One can imagine scenarios that maintain privacy without trust, but such scenarios generally involve only storing data on self-designed, built, and programmed devices that never leave one's possession. Such an arrangement is, at best, impractical for the vast majority of users, and at worst, simply not possible to achieve. The range of manufactures, suppliers, and service providers inherent in the modern computing landscape require that users make decisions regarding whom to trust at every step of any digital interaction in which they partake.

The cloud computing model, by its very nature, further amplifies the number of parties users must trust. But what does it mean to trust a third party in the cloud? Before discussing the details of the SSaaS ecosystem, it is helpful to further define "trust". This chapter defines a model for analyzing trust and applies this model to compare a range of both traditional and privacy-enhancing software and systems.

5.1 Analyses Framework

Generally, when users leverage modern computing devices and services, they must trust third party manufacturers and service providers with their data. The privacy of user data relies on this trust via has two main factor: how much trust must a user place in third parties (e.g. how much of their personal data do they grant the third party access to), and in what manners can the third

party violate this trust (e.g. how can the third party abuse the access they have been granted). A model of third party trust must therefore evaluate trust across two axes: the **degree** of trust users must place in third parties, and the manner in which this trust might be **violated**. The ideal privacy and security enhancing 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 degree of trust, third parties can be trusted 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 necessarily 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 (e.g. changing appointments on a user's calendar, etc).

Meta-analysis (M):

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

Trust violation occurs 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.

There are several types of trust violations. Each is defined by 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 through 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. Another example might include a company going bankrupt and being forced to sell its user data to another entity [286].

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 is a subclass of "Unintentional" trust violation. 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 an employee of a cloud service provider accessing or disclosing private user data without authorization.

Outsider (O):

This is another subclass of "Unintentional" trust violation. This class of violation occurs

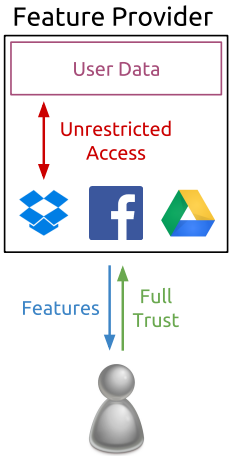


Figure 5.1: Traditional Trust Model

when an external adversary gains unauthorized access to private user data stored by a 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.

5.2 Traditional Model

Existing cloud services are not generally well optimized for minimizing either the degree of third party trust or the likelihood of third party trust violations. Such services tend to apply an all-or-nothing trust model wherein a user must cede a high degree of trust to a service in order to reap the benefits the service provides. Figure 5.1 shows the basic trust-for-features relationship between a user, their private data, and a traditional cloud service provider.

Dropbox (as described in § 1.2) provides a more concrete example of third party trust analysis under this 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. Users therefore grant 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 Dropbox 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. Therefore, Dropbox users must trust Dropbox with all possible capabilities.

But how likely is it that Dropbox might misuse any of these capabilities, thus violating the user's trust? 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. Nonetheless, the user has no way to prevent an **I** violation when using Dropbox, so users can do no more than give Dropbox the benefit of the doubt in this area.

In terms of compelled (**C**) violations, Dropbox is a U.S.-based company, and is thus susceptible to a variety of government-sponsored data requests, from subpoenas issued under the Third Party Doctrine [311], to probable cause search warrants [323], to National Security Letters [80], to Foreign Intelligence Surveillance Court (FISC) [320] orders. Dropbox publishes a transparency report [67] 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 six 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 five hour period [69]. So 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 [68]. 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 demonstrated, a traditional cloud service like Dropbox requires a 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 computing services,

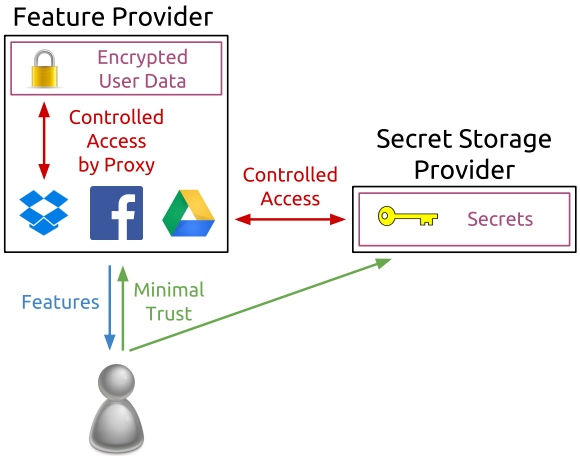


Figure 5.2: SSaaS Trust Model

from social media sites to file lockers to communication systems, all suffer from the same full trust, high violation potential paradigm.

5.3 SSaaS Model

In order to enhance user privacy in the cloud, applications must move beyond the traditional full trust, high violation potential model presented in § 5.2. Between degree of trust and potential for violation, degree of trust is the easier to control quality – what capabilities to trust a cloud provider with is largely within the user's control, whereas the method in which trust might be violated is largely outside of a user's control. As such, the SSaaS model focuses on mitigating degree of trust first while disincentivizing methods of violation second.

The ideal trust model begins with the principle of least privilege [252]: users 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, users must also maintain some degree of auditing over any capability they entrust to a provider. Minimal, audited trust forms the basis of the ideal privacy and security enhancing trust model.

If one applies this ideal trust model to the previous Dropbox example, Dropbox should really only be afforded the storage (**S**) capability. After all, there is no real need for Dropbox to 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 support the desired use case, it is far more difficult for the user to limit this capability than the **R** or **W** capabilities. The user might thus concede this capability to Dropbox as well. In this analysis, Dropbox's required capabilities are reduced from four (full trust) to two (minimal trust).

But how can the user enforce this limited trust profile when using Dropbox? Figure 5.2 shows the proposed SSaaS solution to this problem. The SSaaS model introduce a new actor, the Secret Storage Provider (SSP), into the mix. Using an SSaaS-backed Dropbox application, the user is restricted to only storing encrypted and authenticated data with Dropbox. Such an application then stores the associated encryption and verification keys with the SSP. Assuming the application utilized strong encryption and integrity systems like AES [202] and CMAC [71], Dropbox can neither feasibly decrypt and read user 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 SSaaS-backed Dropbox application can communicate with the SSP from any location where it 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 does the SSaaS model simply replace one trusted third party with another? Why is the SSP any more trustworthy than Dropbox itself? There are several reason why the user might expect an SSP to be less prone to trust violations than a feature provider like Dropbox. (These reasons are discussed in Chapter 6.) But in terms of degrees of trust, the user really isn't affording the SSP any greater degree of trust than 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 raw 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 both of them (**C**-type violation) to provide the data necessary to 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 the trust model:

Colluding (L):

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

This framework provides a basis for analyzing third party trust as well as the basic argument in favor of SSaaS as a trust-reducing system. Chapters 6 and 10 further discusses SSP trust and violation mitigation strategies.

5.4 Trust Survey of Existing Systems

In order to evaluate the trust and security of the SSaaS ecosystem, it is useful to apply this trust model to a number of existing scenarios. Each of these applications is discussed in more depth below and summarized in Figure 5.3.

Table 5.1 shows a summary of the capabilities granted to third parties across a number of common applications. Each application is rated on a four-point scale based on the degree of trust the application requires the user to place in a third party with respect to a given capability. "None" (0 points) indicates that the application requires no third party trust for the given capability. "Minimal" (1 point) indicates that the application makes concerted efforts to minimize the amount of trust third parties require, but where some low level of trust may still be required. "Partial" (2 points) indicates that the application takes steps to minimize third party trust for specific

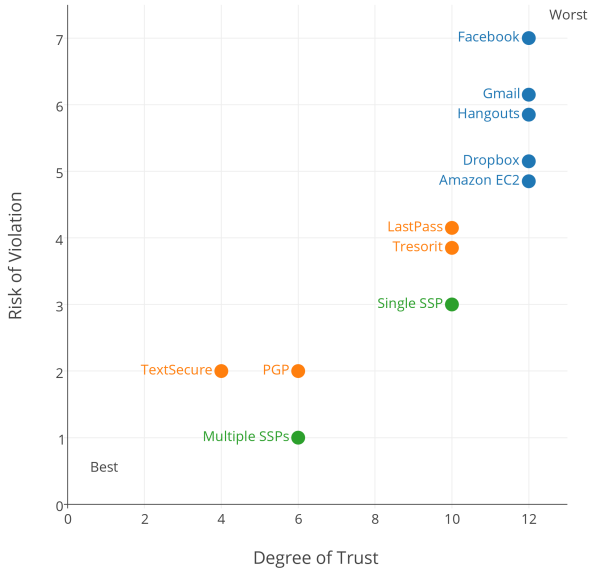


Figure 5.3: Degree of Trust vs Risk of Violation
Traditional Services, *Security-Enhanced Services*, *SSaaS Services*

capacities, but still require more trust than is actually necessary. "Full" (3 points) indicates that the application does nothing to mitigate third party trust for a given capability. The score column shows the point sum of the other columns. Lower scores represent lower degrees of third party trust.

Application	Storage	Access	Manipulation	Meta-analysis	Score
TextSecure	Full	None	None	Minimal	4
PGP/GPG	Full	None	None	Full	6
Multiple SSPs	Partial	Minimal	Minimal	Partial	6
Tresorit	Full	Partial	Partial	Full	10
LastPass	Full	Minimal	Full	Full	10
Single SSP	Full	Partial	Partial	Full	10
Dropbox	Full	Full	Full	Full	12
Facebook	Full	Full	Full	Full	12
Gmail	Full	Full	Full	Full	12
Hangouts	Full	Full	Full	Full	12
Amazon EC2	Full	Full	Full	Full	12

Table 5.1: Degree of Third Party Trust Across Capabilities
[Least Trust] None (0) - Minimal (1) - Partial (2) - Full (3) [Most Trust]

Table 5.2 shows known and likely trust violation risk posed by third parties across the same range of applications. Each application is again rated on a four-point scale based on how likely the third party (or parties) providing a specific application are to commit various types of trust violations. "Insider" and "Outsider" type violations are not included separately since they are really just subclasses of "Unintentional" violation. "Minimized" (0 points) indicates that the third parties backing a given application have taken active steps to minimize the likelihood that a given type of trust violation might occur (e.g. by publicly fighting court orders, undergoing public audits and reviews, etc). "Disincentivized" (1 point) indicates that the third party's business model or other incentivization factors actively disfavor a given class of trust violation. "Vulnerable" (2 points) indicates that a third party has no particular pressure to avoid committing a certain type of trust violation, but that such violations have not necessary occurred. "Known" (3 points) indicates that a third party has a historic record of committing a specific type of trust violation and has not taken steps to reduce the risk of such violations in the future. The score column shows the point sum of the other columns. Lower scores represent less risk of third party trust violations.

Application	Implicit	Compelled	Unintended	Colluding	Score
Multiple SSPs	Disincent.	Minimized	Minimized	Minimized	1
PGP/GPG	Disincent.	Disincent.	Minimized	N/A	2
TextSecure	Disincent.	Disincent.	Minimized	N/A	2
Single SSP	Disincent.	Disincent.	Minimized	Disincent.	3
Tresorit	Disincent.	Vulnerable	Disincent.	N/A	4
LastPass	Disincent.	Vulnerable	Disincent.	N/A	4
Dropbox	Disincent.	Known	Disincent.	N/A	5
Amazon EC2	Disincent.	Known	Disincent.	N/A	5
Gmail	Vulnerable	Known	Disincent.	N/A	6
Hangouts	Vulnerable	Known	Disincent.	N/A	6
Facebook	Known	Known	Disincent.	N/A	7

Table 5.2: Risk of Third Party Trust Violations
[Least Likely] Minimized (0) - Disincentivized (1) - Vulnerable (2) - Known (3) [Most Likely]

5.4.1 Cloud File Storage

As mentioned, cloud file storage is a popular third party use case. Section 5.2 already modeled the trust of Dropbox [65] as an example cloud storage provider. A summary of the Dropbox trust model results are shown the Tables 5.1 and 5.2. As previously concluded, Dropbox both requires a full range of trusted capabilities and is susceptible to (although often disincentivized from) a range of violations of this trust. Google Drive [114], Microsoft OneDrive [189], and related services have similar trust profiles to Dropbox.

As discussed in Chapter 4, a number of systems have been developed with the aim of overcoming third party trust challenges in the storage space. These systems include "end-to-end" encrypted file storage services such as Tresorit [312]. These systems aim to place limits on a third party's ability to leverage the access (**R**) capability through the use of client-side encryption. Likewise, they aim to limit third party access to the manipulation (**W**) capability through the use of client-side data authentication. In the base case where a user merely wishes to store data on a single device and not share it with others, these systems are fairly successful in achieving their desired trust mitigations. In order to sync data across multiple devices using such systems, a user must manually provide some secret (e.g. a password, etc) on each device to bootstrap its operation. While potentially burdensome and inconvenient, this practice is in line with these services trusted capabilities mitigation since it does not require any additional third party trust.

The place where these systems falter at mitigating third party trust is via their support for multi-user sharing and collaboration. Such services tend to accomplish multi-user sharing by acting as a trusted certificate authority (CA) in charge of issuing user certificates.[1] These certificates are then used with various asymmetric cryptographic primitives (e.g. RSA [244]) to exchange the necessary secrets for bootstrapping sharing between users. Unfortunately, as a trusted CA, these services are capable of issuing fraudulent user certificates to themselves or other parties. This allows them to mount man-in-the-middle (MitM) attacks on any user trying to share data by

[1] A certificate is a combination of a user's public key and certain metadata signed by a trusted issuer. See Section 2.1 for more information.

impersonating the recipient of the shared data. This deficiency is discussed in depth at [347], and leads to a breakdown of such services' claim that their users need not trust them, at least when employing multi-user sharing. By mounting a MitM attack on a user trying to share data with another user, such service providers can regain the **R** and **W** capabilities they claim not to have. Furthermore, these services do little to mitigate their access to metadata (**M** capability). Nor do they provide ways for users to avoid data loss in the event that one of the services goes offline or shuts down (**S** capability).

In terms of trust violations, such "secure" cloud storage services are being paid to store user data, disincentivizing implicit violations. Compelled violations are possible, although as services that market themselves as "privacy" products, compelled violations (assuming they are public) are also disincentivized. Due to the CA trust requirements that exists in such services' sharing implementations, however, it is possible that such a service could be compelled to mount a MitM attack on one of their users in order to provide such data to the compelling party (similar to government efforts to compel Apple to create a flawed version of iOS that would be vulnerable to brute force attacks [15]). Similarly, such a service could be compelled to surrender their CA private key, allowing the compelling party to take over the trusted CA role and mount such a MitM attack themselves (similar to the how Lavabit was compelled to turn over their private TLS keys to facilitate government access to the data they controlled [171]). The fact that such capabilities have been exploited for the purpose of committing compelled violations in the past raises the likelihood of such violations when using services such as Tresorit in the future. Unintentional, Insider, and Outsider violations are all similar to the traditional use case: such violations are technically possible, but the third party has a vested interest in avoiding such violations for reputation-related reasons. Collusion violations aren't really applicable since Tresorit is a single-party actor.

Furthermore, since services such as Tresorit are not open source or widely audited by independent parties, the user must trust that the code they are running to access these services is faithfully implementing the security guarantees the services claim to support. It is possible that

such a service could ship a flawed version of their code due to a wide range of trust violations (e.g. they could be compelled to ship such code, or it could be an honest unintentional mistake).

A summary of the Tresorit trust model results are shown the Tables 5.1 and 5.2. Such a service does more to mitigate third party trust and reduce the likelihood of violations than a traditional cloud storage service, but still leaves room for improvement. SpiderOak [288], Wuala [163], and related services have similar trust profiles to Tresorit.

5.4.2 Social Media

Social media sites such as Facebook, Google Plus, etc have become popular since the early 2000s. Such sites maintain a "social-graph" of connections between users, and facilitate communication and sharing of pictures, events, and other data between users. Such sites are generally "free" to users – monetizing user data and interactions for the purpose of selling targeted advertising. Given their ubiquity in the modern Internet landscape, as well as their position as ad-supported services, it is useful to evaluate the trust profile of modern social media sites. Facebook is the largest social media site today, serving over 1.5 billion users as of 2015 [87]. As such, Facebook serves as an example of the variety of social media sites available today.

In terms of capabilities, Facebook, like Dropbox and other traditional cloud services, must be trusted with a full range of capabilities. Facebook is responsible for faithfully storing user data, Facebook can access and read all data it stores, Facebook can manipulate the data it stores, and Facebook is capable of performing meta-analysis of what data users are accessing and of how they are accessing it.

In terms of likelihood of trust violations, Facebook differs a bit from cloud-based system like Dropbox in that it is an ad-supported system. This increases the likelihood of implicit trust failures due to Facebook's business model being based around the sale of user data. Indeed, and as mentioned in Chapter 3, Facebook has a record of implicit trust violations ranging from the Emotional Contagion Study [102] to its use of users' photos in the ads it serves [341]. Beyond implicit trust violations, Facebook's trust violation profile is again similar to other traditional

cloud services such as Dropbox. Like Dropbox [67], compelled violations are known to occur [77]. Unintentional, insider, and outsider violations are all possible, but Facebook has an active interest in avoiding them. Collusion violations aren't really applicable since Facebook is a single-party actor.

Tables 5.1 and 5.2 show a summary of the Facebook trust model analysis. Google Plus [120] and related social media services have similar trust profiles to Facebook.

5.4.3 Communications

Communication systems ranging from email and chat to voice and video calling are another popular use case. The privacy and security of these systems are a matter of great public concern, and indeed many of the current privacy and security related legal battles revolve around the ability to communicates in a private and secure manner (e.g. [51, 125, 171]). It is useful to analyze the trust of several such services. To illustrate a range of profiles, it is useful to analyze both traditional (Gmail) and secure (GPG/PGP) email services, as well as traditional (Google Hangouts) and secure (TextSecure) chat/messaging applications. Tables 5.1 and 5.2 show a summary of these profiles.

Gmail [116] and Hangouts [119] are both Google-hosted communication services, placing them fairly soundly in the "traditional cloud services" camp. As such, their profiles are similar to those of Dropbox and Facebook. Both services require granting Google a full range of capabilities over user data.

Like Facebook, both services are ad-supported, increasing the likelihood of implicit trust violations (although unlike Facebook, Google seems to have a better track record of not committing such violations). Similarly, both services are known to be subject to compelled violations [118],[2] and are disincentivized, although by no means immune, from committing unintentional, insider, and outsider violations.[3] Thus, Google is able to analyze, alter, and expose any data that travels

[2] It is possible such cloud communication services are actually at an increased risk of compelled violations relative to other cloud services due to the existence of laws such as the Communications Assistance for Law Enforcement Act (CALEA) [318, 334] specifically designed to aid the government in obtaining a user's communications.

[3] Google has taken steps to ensure communication between all of its data centers and between mail servers are encrypted in transit, helping to minimize outsider violations [173]. Similarly, it has recently started providing end users with information as to whether or not encrypted email transmission was possible and whether or not an email's

across its network using either service – either of its own volition or due to a compelled order or unauthorized data breach.

The OpenPGP protocol [44] is one of the traditional mechanisms for securing digital communications. It is most widely used with email and is implemented via a variety of applications from desktop apps such as Enigmail [75] (powered by GnuPG (GPG) [153]) to browser-based plugins such as Mailvelope[182]. The OpenPGP protocol utilizes public-key cryptography to end-to-end encrypt and authenticate messages traveling across untrusted mediums. It can be applied atop mail traversing traditional email systems such as Gmail, as well as to messages traversing chat applications such as Google Hangouts. When used with such service, PGP provides an additional level of trust mitigation above and beyond what is possible to achieve via the native services themselves. In terms of trusted capabilities, a user employing PGP atop a traditional third party cloud service such as Gmail minimizes both the third party's access (via encryption) and manipulation (via authentication) capabilities. In such a scenario, only the end users involved in a given communication, and not any third party through which that communication might pass, have access to the necessary cryptographic keys required to read or alter the message. The third party, however, can still capture metadata about the communication since such metadata is outside of the scope of the message content that PGP is capable of securing. The third party is also still capable of dropping or deleting the communication all together, and thus still possesses the "storage" capability.

Since the OpenPGP protocol and associated implementations are not controlled by any single third party, there is no straightforward analysis of its likelihood of PGP-related trust violations. In general, however, PGP implementations are controlled by parties with a vested interest in maintaining user security and privacy. Such parties are strongly disincentivized from committing implicit or compelled violations. Most OpenPGP implementations are also open source. This helps to mitigate[4] unintentional, insider, and outsider violations by maximizing the number of eyes on

sender has been authenticated [237]. In both cases however, third parties, not end users, remain in full control of all the necessary cryptographic keys, limiting this protection to the mitigation of outsider violations, and doing little to mitigate Implicit, Compelled, Unintentional, or Insider violations.

[4] Although it does not necessarily prevent such violations, e.g. as in Heartbleed [48] or a Ken Thompson style attack [310].

the code and reducing the likelihood of mistakes or intentionally coded vulnerabilities and back doors.

Due to the numerous challenges and deficiencies associated with using OpenPGP-based systems [123], developers have created a number of alternate secure communication protocols. These protocols aim to provide forward-secrecy, metadata privacy, deniability, contact authentication, and message encryption and authentication for (primarily) real-time communication such as instant messaging and chat systems. Examples of such protocol include OTR [223] and OTR-derived protocols like TextSecure [184]. The TextSecure protocol is used by several apps such Open Whisper System's Signal [215] and WhatsApp [343]. As such, TextSecure is thus a useful target for trust profile analysis.

TextSecure uses various forms of asymmetric cryptography to provide users with end-to-end encrypted and authenticated individual and group messaging capabilities. TextSecure has been formally analyzed and proven to provide secure encryption and message authentication [90]. Use of TextSecure denies any third party through which TextSecure messages might pass (including the TextSecure server itself) the access or manipulation capabilities. Furthermore, TextSecure makes efforts to secure all metadata from third party actors, including the server itself. This strongly curtails a third party's ability to analyze message metadata. It is still possible for a network-level adversary or the TextSecure server to discover the raw network (e.g. IP) endpoints involved in a TextSecure exchange, but higher level details are not available. It is also possible to couple TextSecure with existing network anonymity systems such as Tor [63] to mitigate such network-level meta-analysis [169]. TextSecure users are still dependent on a third party to operate a TextSecure server in order to communicate in the first place (e.g. it is not a fully distributed protocol), but beyond this "storage"-like capability, TextSecure grants no other capabilities to any third party.

Like OpenPGP, TextSecure is a protocol implemented by several different apps. Such application providers (e.g. Open Whisper Systems) specifically market their products as being a secure alternative to more traditional chat systems, and are thus strongly dissuaded from committing any kind of implicit or compelled trust violation. Furthermore, most TextSecure implementations

are open source, which mitigates the likelihood of unintentional,[5] insider, or outsider violations. TextSecure has also undergone external audits and reviews, further decreasing the likelihood of an unintentional or insider violation [90]. As in OpenPGP, even if a violation occurred, the capability restrictions discussed above severely limit what data the violation would expose. Therefore, TextSecure represents a successful effort to reduce third party trust exposure and secure user communications.

5.4.4 Password Managers

Password management programs are commonly used by end users wishing to both manage and increase the security of the credentials they use to access web-based resources. Such programs are useful for helping end users remember passwords, and by extension, for encouraging users to use stronger (i.e. longer and/or more random) passwords [41, 159, 270]. It is useful to evaluate the trust profile surrounding such programs. LastPass [164] is one of the most popular cloud-based password managers. The results of evaluating LastPass's trust profile are summarized in Tables 5.1 and 5.2. Other cloud-based password managers (e.g. 1Password [5]) have similar profiles.

LastPass operates by storing encrypted user passwords on a LastPass-controlled server. Passwords are enciphered on the client and only encrypted passwords are sent to LastPass. Each password is encrypted using a key derived from a user-supplied "master" password. LastPass never stores this master password directly, making it difficult for them to derive the key necessary to decrypt the encrypted data they store. Thus, LastPass limits its "access" to user data. LastPass does not, however, appear to perform any kind of cryptographic authentication on the data it stores, meaning it still has full capabilities over the "manipulation" capability.[6] Similarly, LastPass is fully responsible for faithfully storing user data and has full access to all user metadata associated with any stored password.

[5] Researchers have discovered bugs in TextSecure, but these bugs were quickly patched [90].

[6] In realty, it is somewhat difficult for LastPass to modify user data intelligently since they lack the ability to read the result of their modifications. Nonetheless, the lack of any client-side cryptographic protections against modification leaves the door open to a range of potential attack on LastPass's client side encryption as per the "cryptographic doom" principle [183].

In terms of likelihood of trust violations, LastPass has a very similar profile to a service like Tresorit. LastPass is a "freemium" service that generates its income off its ability to faithfully store and protect user passwords, encouraging users to pay for higher level service tiers. This disincentivizes implicit trust violations. LastPass also has a strong incentive to minimize (at least public) compelled violations, although it is still subject to such violations. Unintentional, insider, and outsider violations are similarly disincentivized, although they have occurred before [278].

Like Tresorit, LastPass is neither open source[7] nor subject to public code audits. As such, the user is required to trust that LastPass has faithfully coded its software not to expose additional capabilities via hidden back doors. It is possible that LastPass could be compelled to modify the LastPass client program to do something like send copies of a user's master password back to LastPass where it could be used to decrypt all of the user's data. It is also possible that the lack of cryptographic authentication on user data would allow LastPass to mount a range of attacks on a user - potentially revealing one or more of the passwords they store with LastPass in the process.

5.4.5 Cloud Infrastructure Services

Most modern cloud services are operated atop infrastructure provided by shared infrastructure service providers such as Amazon or Google. The benefits of such arrangements are discussed in detail in Section 2.5. Given that cloud infrastructure is the primary method of deploying most modern software, it is useful to analyze the trust a user of such a service must place in their provider. Amazon's Elastic Cloud Compute (EC2) service is a typical Infrastructure-as-a-Service (IaaS) example. A summary of EC2 trust profile results appears in Tables 5.1 and 5.2.

Amazon EC2 [10] allows a user to spin up user-controlled virtual machines (VMs) running atop Amazon-owner servers. In such a situation, a user can control the specification of their VM, but is entirely reliant on Amazon to faithfully provide the underlying hardware. The traditional wisdom in computer security is that anyone who controls the hardware has the ability to subvert

[7] Although parts of LastPass are build using JavaScript, which by its very nature exposes the source code to users. Nonetheless, such code can be obfuscated to make it difficult to read, and users lack most of the "Free Software" rights [89] traditionally associated with "open source" code.

any software-level security control implemented atop it. Since Amazon controls the hardware underlying EC2, they effectively have full control over any data that end users store atop EC2. Amazon could (theoretically) read all a VM's memory or disks, or even alter the behavior of the processor in such a system to control the execution flow of any software running atop it. As such, EC2 requires the end user to grant Amazon full capabilities to store, read, modify, or meta-analyze any the data stored or processed via an EC2-backed VM.[8]

The profile of Amazon's proclivity for trust violation follows that of other service-oriented cloud operators such as Dropbox and LastPass. Amazon charges users for the right to use EC2, and faces competition from numerous other actors in the cloud service provider space. Thus, Amazon is strongly disincentivized from committing any form of implicit trust violations. Amazon, like other third parties, is subject to a range of compelled violations [8]. Likewise, Amazon is disincentivized, but not immune to, unintentional insider or outsider violations.

5.4.6 SSaaS Alternatives

All of the applications analyzed thus far have employed various non-SSaaS-based models for security and secret management. For comparison, it is useful to analyze two of the primary Secret Storage as a Service models discussed in Chapter 6: the single secret store provider (SSP) model and the multi secret storage provider model.

In the single SSP model, the end user stores their secrets with a single SSP third party, potentially using these secrets to protect a separate set of data stored with traditional cloud provider. Such a model is used by the Custos prototype presented in Chapter 8. In such a model, the end user still grants one or more third parties the full "storage" capability since the failure of either the SSP or the traditional cloud provider will result in a loss of user data. In such a scenario, however, the end user is granting a more limited set of "access" and "manipulation" capabilities since no single third party has access to both a user's data and the secrets necessary to decrypt or alter it.

[8] It is potentially possible to use encryption system that don't require ever storing the encryption key itself atop Amazon-controlled infrastructure to work around some of these issues, but such homomorphic encryption systems are beyond the standard EC2 usage base case discussed here. See Section 4.2 for more details.

As described in Section 5.3, the user can store encrypted and cryptographically authenticated data atop Dropbox and store the associated cryptographic keys with their SSP, ensuring that neither the SSP nor Dropbox can easily read or manipulate their data. The SSP can, however, modify and read the user's secrets themselves, which, depending on the use case, may still allow them to access a variety of user data depending on the nature of the SSP-backed application in question. Similar to previously discussed systems, the single-SSP model still allows both the SSP and any associated cloud storage providers a high degree of access to user metadata regarding the secrets they store or the data such secrets exist to protect.

In terms of likelihood of trust violations, the single-SSP model is similar to other privacy-preserving applications such as TextSecure [90]. In a healthy SSaaS ecosystem, there should be a multitude of competing SSPs, one or more of which an end user will pay for secret storage. This competition, coupled with the storage-for-profit business case, disincentivizes implicit trust failures by any single SSP. Compelled violations are similarly disincentivized. Indeed, it is likely that an SSP would take steps to minimize compelled failures in order to uphold its reputation. SSPs can also mitigate unintended violations by ensuring they use publicly audited and/or open source implementation of their systems. A single SSPs is, however, at risk of colluding with whichever storage provider a user is using to store their encrypted data. If such an SSP colluded with such a storage provider, both parties could succeed in accessing the user's data and regaining the "access" and "manipulation" capabilities they claim to avoid.

In order to improve upon the trust profiles inherent in the single-SSP model, it is possible for user to move to a multi-SSP SSaaS model. In such a model, the user shards their secrets across multiple SSPs using a n choose k scheme that provides both privacy and redundancy (details of such schemes are discussed in more depth in Section 6.3). The multi-SSP model requires a reduced degree of trust compared to the single SSP model. In the multi-SSP model, the user can mitigate the need to trust any single third party with the "storage" capability. Instead, a user can redundantly spread their data across multiple SSPs. Assuming a user selects $n > K$, the failure of one or more SSPs will not result in data loss. Using multiple SSPs also further reduces each SSP's "access" and

"manipulation" capabilities since no single SSP has access to the user's complete secret. Finally, the use of multiple SSPs may even reduce the user's metadata exposure by allowing the user to avoid having any single SSP hold all of their secrets or see all of their secret access patterns.

The multi-SSP model also reduces the risk profile associated with SSP trust violations. As in the single SSP case, implicit violations are disincentivized by nature of the SSP business model. In the multi-SSP use case, however, compelled violations are further minimized since no single SSP can be compelled to provide all of the required data to access a user's secrets. As such, a compelling entity would be forced to compel multiple parties, potentially located across multiple jurisdictions, to turn over user data in order to access a user's secrets. Unintended violations are again minimized assuming prudent use of open source code and public audits. The risk of collusion-related violation are also further reduced due to the fact that the multi-SSP use case requires a greater number of colluding parties to succeed in mounting a collusion-type violation. The greater the number of required colluding parties, the less likely such collusion is to occur.

Tables 5.1 and 5.2 provide a summary of both the single SSP and multi-SSP SSaaS models. As these tables show, the trust profiles of such models score on par or better than existing privacy-enhancing systems such as PGP and Tresorit. As discussed in later chapters, however, such models offer additional functionality above and beyond that achievable using other security and privacy enhancing solutions.

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 system that increases the security and privacy of users by reducing their exposure to third party trust abuses. Such mechanisms also provide additional security beyond third party risk, e.g. by ensuring that systems such as mobile devices remain secure even if lost or stolen. Unfortunately, cryptography is not merely "magic fairy dust" that developers can sprinkle on any security or privacy problem to make it disappear [284]. Effectively using cryptographic technique to secure user data involves ensuring that cryptography-based solution are designed in both a secure and a usable manner.

One of the critical components of building secure and usable cryptographic data security solutions lies in providing secure and flexible key management systems that can be leveraged to store the cryptographic keys associated with such systems. 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. Such key management systems are but one facet of the larger secret storage problem: the need to use to store a range of secrets from passwords to personal data in a manner that is both secure and usable.

This chapter discusses the Secret Storage as a Service (SSaaS) model: a service oriented secret storage method designed to provide users with the necessary tools for managing secrets in a

manner that allows for a range of use cases while also avoiding the need to place high degrees of trust in any single third party.

6.1 Architecture

Secret Storage as a Service (SSaaS) is a service oriented architecture where users utilize dedicated Secret Storage Providers (SSPs) in addition to traditional Feature Providers (FPs) such as 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. FPs, on the other hand are limited to storing less sensitive information such as cryptographically protected data,[1] the keys for which reside with one or more 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 no single entity need be fully trusted, while still enabling many popular use cases.

6.1.1 Stored Secrets

In the SSaaS model, what kind of secrets does a user store with an SSP? 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 the secrets stored are not themselves inherently sensitive or revealing. This property helps to mitigate the amount a user must trust each SSP. Thus, storing secrets like cryptographic keys that alone reveal no private user data is generally preferable to storing privacy revealing secrets like social security numbers, etc. The decision of what to store with each SSP, however, is left up to each user and application to maximize the flexibility of the SSaaS model. Chapter 7 explores various types of secret storage applications.

Another consideration related to what secrets to store with an SSP is size. It is reasonable to predict that SSP-based storage will sell at a premium price vs more traditional cloud storage

[1] At least in the normative case. There are also SSP use cases that involve no FP at all, e.g. storing user passwords.

options like Amazon S3 [11]. 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 or a greater emphasis of resistance to compelled violations via legal representation. Thus, it may be desirable for the user to minimize the amount of data stored with an SSP as a cost optimization. Such optimizations make use cases such as storing small cryptographic keys with an SSP while storing larger encrypted data with a more traditional provider desirable.[2]

6.1.2 Secret Storage Providers

In the SSaaS model, SSPs will offer a standard set of features. These include a standardized storage 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 (i.e. the "value") is tagged with a unique identifier (i.e. the "key").[3] The secret associated with each ID could be a cryptographic key, personal user data, or any other arbitrary secret value. Users are able to query each SSP for the secret associated with a given ID.

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" [142] capabilities can be built atop versioning schemes that maintain access control information on a per-version basis.

[2] For these reasons, 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.

[3] Most likely a UUID [166] or similar unique ID standard.

Each SSP will expose its key-value secret store via a RESTful HTTPS-based API.[4] 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, it is likely that most SSP implementations will use an off-the-shelf key-value store as the backend for storing user secrets. SSPs should utilize a standard API in order to allow users 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 creator of each secret, potentially delegating such control to other users as well. 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). It is desirable for SSPs to offer a standard access control framework in order to promote interoperability between multiple SSPs.

It is 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 require autonomous access to a secret used by an

[4] The exact mechanism powering the API interface is not relevant to the SSaaS model as a whole, but given the ubiquity and simplicity of RESTful designs, it seems a reasonable choice for an SSaaS standard.

automated processes (e.g. a secure backup system), potentially limited to systems possessing a special token or to specific times of day.

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 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 its 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 is 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. For example, if User A shares a secret with User B by granting them read access via their SSP, but then decides they would rather revoke that access, User A can check the audit logs to determine if User B has already accessed the shared secret and thus whether or not guaranteed revocation is even possible.

As with the other SSP core functions, an SSP's auditing capabilities will be exposed via the SSP API 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 [36] or [165]. Alternatively, such systems might be constructed using block-chain-based primitives such as those available in the BitCoin crypto-currency network [200]. Such public audit logs might provide more robust variants of the "warrant cannery" concept that has recently become popular amongst a range of third party services providers as a counter measure against secret compelled data requests [221].

SSP audit logs could also be interfaced with third party analysis engines (e.g. [175] or [289]) in order to spot abnormal access patterns or other security-related anomalies. The logically-centralized nature of SSPs make them a desirable point at which to audit and detect unauthorized access requests for user data via such mechanisms.

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 7 discusses potential SSaaS use cases and applications in detail. This section outlines some of the basics of SSaaS client design and operation.

An SSaaS client is any system designed to connect to and utilize a Secret Storage Provider. Clients communicate with one or more SSPs via the SSP API. Clients can store and retrieve secrets with each SSP, manage 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 service is a good candidate for integration into an SSaaS architecture. Such benefits might be derived either 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 from the simple desire to avoid implementing a full secret management and access control stack locally,

In the simplest case, each SSaaS client communicates with a single upstream SSP via a standard 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 a single SSP arrangements lies in the fact that storing each secret with only one SSP raises both trust and availability concerns (the details of which are discussed below). To overcome such concerns, it is important for each SSaaS client to be able interact with multiple SSPs, sharding each secret across multiple SSPs and increasing reliability while reducing trust in the process. Such multi-SSP arrangements, however, raise client side management issues. How are access control rules managed across SSPs? How does the client keep multiple SSPs in sync? Potential answers to these questions are provided by the Tutamen work presented in Chapter 9.

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 services provide. Privacy and security, while potential concerns, are at best secondary goals. Furthermore, on many free cloud services, the ability to harvest user data is the basis of the service provider's business model. As discussed in Section 5.4, these situations create a number of perverse incentives in terms of a traditional feature provider's goals with respect to user security and privacy [13]. In the first case, the feature provider simply does not prioritize user security since that is not the primary basis on which users are choosing to pay for a service. In the second case, a feature provider actively works 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. 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, optimized for the respective priorities of each field.

Beyond the removal of perverse incentives brought about by the separation of SSPs and FPs, it is also desirable to encourage a competitive market amongst multiple SSP providers. In order to achieve such a market, it is necessary to standardize on 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 – a practice employed by many traditional feature providers.[5]

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. Such sharding (discussed below) provides a number of benefits over relying on a single SSP, from trust reduction to data redundancy.

[5] As an example, consider Facebook's numerous trust violations [102, 176, 314] and the fact that such violations have had no noticeable impact on the number of people using Facebook [87]. An SSP would enjoy no such network benefit from providing additional services beyond secret storage were they to violate user's trust; instead, users would simply switch to a new SSP.

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 liability and 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. SSPs could provide insurance policies to their users to indemnify them against any loss caused by a trust violation on the part of the SSP. Likewise, SSPs would underwrite such user-facing insurance policies with their own insurance policies provided by independent third party insurers. These insurers would need to perform independent audits[6] of SSP infrastructure and policies in order to evaluate trust violation risk. Such audits further incentivize SSPs to design their services specifically for the avoidance of such trust violations. Such "cyber" insurance benefits are not as readily available in a mixed trust + feature cloud ecosystem since it is far harder to evaluate the privacy violation risk of a company whose primary objectives are more complex then secret storage alone [213]. The increasing regulation of user privacy rights and the penalties associated with violating these rights further incentivize 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. These ideas are discussed in more depth in Chapter 10.

Likewise, SSaaS provides compliance benefits to users storing highly regulated data by shifting compliance burdens from FPs to SSPs. Instead of having to individually verify that each FP cloud service meets the requirements of a specific data storage regulation, a user could instead simply make sure that their SSP meets the necessary requirements. Once certified, a single SSP could be reused with multiple FPs without having to 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 store encrypted data with. Such practices would likely prove beneficial in highly regulated fields such

[6] Potentially similar to the audit model used by Certificate Authorities in the PKI system, e.g. [128, 198].

as health care (e.g. requiring HIPPA [331] compliance), education (e.g. requiring FERPA [330] compliance), and online payment processing (e.g. requiring PCI DSS [228] 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, a user wishes to store unencrypted data directly with an SSP,[7] they must place a higher degree of trust in an SSP. In the split encrypted data + cryptographic secrets use case, the user must only trust the SSP with the storage (**S**) and metadata (**M**) capabilities, the same capabilities with which they must trust the FP. But in the case where an SSP directly stores raw user data, they 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, SSPs would appear 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, whereas traditional FPs often have economic incentives in direct conflict with upholding user trust. This fact alone decreases an SSP's 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.

Nonetheless, it is best if users can avoid placing high degrees of trust in any single third party, FP or SSP. Fortunately, there is a viable alternative for avoiding placing a high degree of trust in an single SSP, even in the raw-secret storage SSP 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 [274] ensure that a secret split into n shares can not be reassembled using fewer than k shares in an unconditionally secure manner. Such schemes can be applied in an SSaaS

[7] Potentially because a given use case can't easily be split into encrypted data and encryption keys.

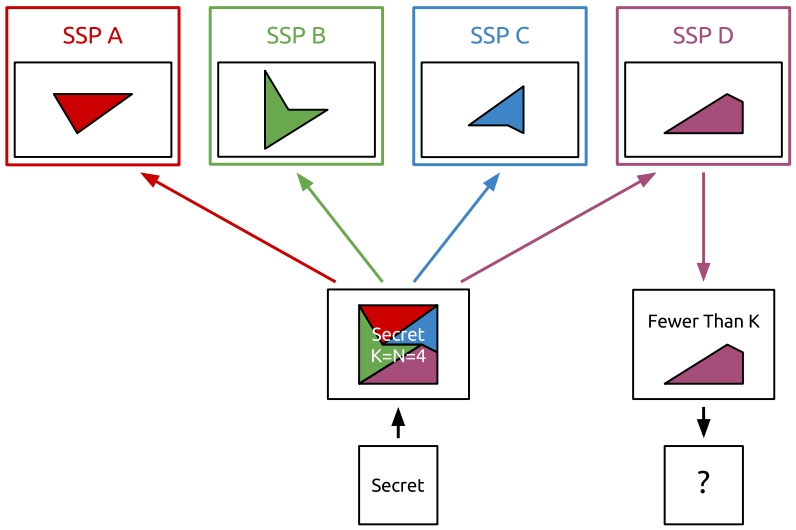


Figure 6.1: Sharding Secrets Between Multiple Providers

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 ensures 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, sharding secrets, be they cryptographic keys or raw data, across multiple SSPs is likely to become a standard best-practice in SSaaS ecosystems for both the trust reduction and increased reliability such a practice provides.

Even with secret sharding, 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, users should select a set of SSPs across which to shard their secrets that minimize **L** and **C** type violation risk. Thus, in addition to the reputation-based selection criteria outlined in § 6.2, SSaaS user's should consider several additional SSP selection criteria:

Geopolitical Diversity: An ideal set of SSPs will 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. See Chapter 10 for additional discussion of this point.

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 shards their secrets across a diverse set of SSPs, all with strong privacy-preserving reputations, they have a reasonable likelihood of avoiding a series of trust violations resulting in the compromise of their privacy or security.

Chapter 7

SSaaS Applications

As mentioned in previous chapters, many applications can utilize the SSaaS model to allow users to reduce the trust they must place in any single third party while still supporting popular use cases. This chapter presents a discussion of common SSaaS application patterns as well as a description of several specific SSaaS applications.

7.1 Common Patterns and Challenges

There are a number of common patterns and challenges that occur when building SSaaS applications. This section discusses some of the more prominent issues faced by SSaaS applications and proposed solutions to each.

7.1.1 SSaaS Metadata Storage

Many SSaaS applications need to store metadata related to their SSaaS-based operations. For example, "key storage as a service" (KSaaS) applications (e.g. client-side encryption, etc) require a mechanism for mapping encrypted data to the key required to decrypt it. Indeed, most SSaaS applications need a way to store useful identifiers for each secret in order to facilitate the identification of secrets by end users (e.g. a password manager needs to map services to the corresponding secret/password). SSaaS applications must also store the identity of the SSP (or SSPs) the user has selected for the storage of their secrets in order to retrieve those secrets later.

Applications must decide how to best to store SSaaS-related metadata in a manner that allows it to be quickly retrieved for the purpose of identifying or accessing specific secrets. The need to maintain such metadata is further complicated by the desire to support secret sharing and syncing use cases. Both sharing and syncing operations require the ability to share or sync the associated secret metadata in addition to the secret itself. This requirement restricts the options for storing metadata to designs that support sharing and syncing.

KSaaS-based encryption applications serve as a good example of the kind of metadata SSaaS applications must maintain. Such split data/key applications require the ability to map encrypted data to encryption keys in a manner that allows the required key to be identified during the decryption process.[1] This mapping represents an additional layer of persistent state that an application must maintain. The loss of such a mapping would render any data stored by the application useless since the application would no longer be able to decrypt the data. It is worth noting that such key-mapping lookups are predominately unidirectional: applications needing to lookup the key corresponding to a known piece of data are far more common than applications needing to lookup the data encrypted with a known key. This is due to the fact that most applications (and their users), natively operate in terms of data objects (e.g. files, photos, etc), hiding the encryption from the user to increase usability. It is far more natural for a user to request access to a piece of data for which the application must lookup the corresponding key than it is for a user to specify a key and need the applications to lookup the corresponding piece of data.[2]

There are essentially three options for storing SSaaS metadata such as that required to map keys to data (or vice versa). Each has pros and cons. It is possible that applications may wish to combine several techniques for the purpose of leveraging the strengths of each while avoiding the flaws of any.

[1] This description uses encryption as an example of a split data/key SSaaS use case, but these ideas are applicable to data authentication use cases and other SSaaS-backed cryptographic primitives as well.

[2] The main exception to this rule is potential SSaaS management applications where a user may in fact wish to lookup a specific key and wish to ascertain which data it is used to protect.

Out-of-Band Metadata: An application may maintain an out-of-band mechanism for storing metadata. For example, an application could maintain a local database of data:key pairs. The advantages of such a solution are in its simplicity and ability to access metadata without needing to perform any form of remote API call to retrieve the corresponding secret or data. The ability to perform such local lookups may have performance benefits for certain applications. The biggest downside to this approach is that it requires the use of additional out-of-band mechanisms to share metadata with other users or sync it with other application instances. The need to implement such mechanisms increases the complexity of SSaaS applications. This approach also has the downside of increasing the risk of metadata loss. Storing the metadata via a separate mechanism unrelated to either the data or secret storage mechanism to which the metadata relates introduces an additional point of failure. As mentioned, the loss of metadata such as data-to-key mappings risks render both the corresponding data and keys meaningless.

Key-Tied Metadata: An alternative to maintaining out-of-band metadata storage is to store all required metadata via the SSP secret storage system. Such storage can be accomplished by allowing the application to provided arbitrary metadata fields associated with each secret or by allowing the application to encode metadata into the secret ID itself. For example, an application could store the identity of the data to which a given key corresponds as a metadata attribute of the key itself. This has the benefit of avoiding the need for an out-of-band database, and has a graceful failure mode in which metadata is only lost if the secret to which it corresponds is also lost – in which case the metadata would likely be useless anyway. The main downside to this approach is that it is not well optimized for computing data-to-metadata lookups such as those required to discover the ID of the decryption key for a piece of encrypted data. Such lookups require an inefficient exhaustive search of all secrets until the one with the required metadata value is found.

Data-Tied Metadata: An application can also invert the idea of a key-tied metadata by storing all SSaaS metadata with the data itself. This might be accomplished either by leveraging

data-linked metadata fields (e.g. the extended attributes offered by many file systems) or by writing additional data to the storage medium itself (e.g. using a "header" structure that precedes each chunk of encrypted data and identifies the key required to decrypt the data). Like key-tied metadata, data-tied metadata avoids creating an additional point of failure by linking metadata storage to the data that depends on it. Furthermore, data-tied metadata can perform constant-time data-to-metadata lookups. This makes such techniques well optimized for encryption systems requiring quick data-to-key lookups. Furthermore, such metadata storage adapts well to sharing and syncing use cases since any mechanisms already in place to share or sync the underlying data (e.g. Dropbox) will also share and sync the associated SSaaS metadata. The main downside to data-tied metadata storage is the need to perform an exhaustive search to compute key-to-metadata lookups. But since such lookups tend to be rare, this deficiency may not be widely noticed.

In many SSaaS applications, it is likely desirable to use a combination of these techniques. For example, an application could use both key-tied and data-tied metadata storage to ensure both constant time data-to-key as well as constant time and key-to-data lookups. Furthermore, an application could also use a local metadata database as a high speed cache for common metadata lookups, avoiding the need to query the data storage medium or SSP to access metadata in many instances. The downside to a combined approach is the additional complexity required keep the multiple metadata stores internally consistent. Such challenges, however, are common in data storage applications and have a variety of general purpose solutions.

7.1.2 Data Granularity

In addition to metadata storage for maintaining data-to-key mappings, encryption and authentication based SSaaS applications must also tackle the decision of at which granularity they wish to encrypt or authenticate data. Traditionally this decision is driven by complexity, performance, and access control requirements [172]. SSaaS-backed applications are no exception.

As an example, consider a disk encryption system. This system could encrypt data in a variety of ways: it could encrypt individual disk blocks, it could encrypt individual files, or it could encrypt individual directories and all the files they contain. Encrypting individual disk blocks using a single key tends to be fast and simple, but makes controlling access to specific files, as opposed to the entire disk, difficult. The higher-level file and directory encryption provides finer grain control over access (e.g. by using separate encryption keys for each file), but incurs additional file system overhead. In general, lower level mappings are going to be faster, but lack support for finer grained access control. Higher level systems are slower, but have more flexible access control options.

The SSaaS model complicates this further by adding additional overhead to the key retrieval phase of any encryption or data authentication operation. Thus, a block-level disk encryption system that uses a single key to protect an entire disk incurs minimal SSaaS overhead since it requires only a single SSP key lookup when the disk is first accessed. But such a setup would limit the user to a single set of SSP-enforced access control rules for the entire disk. Per-file encryption, on the other hand, incurs significantly more (but still manageable) overhead by required an SSP key lookup for each file open operation. File mappings, however, allow the user to set individual access control rules for each file.

There exist additional encryption granularity options. For example, the SSaaS model could be used to store a separate key for each individual hard disk sector. Such a setup would allow potentially interesting methods of per-sector disk access control, but would also significantly increase disk latency by requiring a round trip SSP key lookup for each individual sector. Such a design could incur thousands of lookups to access a single file. Developers must consider the additional SSP-overhead the SSaaS model entails when deciding on the level of granularity required for a given encryption application.

7.2 Use of SSaaS Libraries and Utility Programs

One of the benefits of the SSaaS model is the use of a standardized protocol providing access to secret storage functionality. As mentioned in Chapter 6, this standardization has numerous benefits,

most notably the encouragement of SSP competition. An additional benefit of standardization is the ability to use standard SSaaS libraries and utilities across a range of SSaaS applications. Such systems reduce the complexity of each individual SSaaS application by allowing common SSaaS tasks to be abstracted into dedicated libraries or utilities.

The implementation of SSaaS client functionally will often be non trivial, especially when considering the need for such applications to support the ability to shard secrets across multiple SSPs. The potential complexity of such solutions likely calls for the creation of a standardized SSaaS client libraries that can be utilized by multiple applications. Such a library avoids the need for each SSaaS application to reimplement SSaaS API communication primitives directly. Similar to the manner in which most applications use common cryptography libraries to avoid needing to reimplement cryptographic code, SSaaS applications can use common SSaaS libraries to simplify their integration with the SSaaS ecosystem.

Furthermore, it may also be desirable to create common SSaaS management utilities. Many of the SSaaS-related management features – e.g. setting up access control requirements, checking audit logs, etc – are common across all SSaaS applications. It is thus desirable to offload such functionality to a general SSaaS management utility in cases where such functions need not be tightly coupled with a given SSaaS-backed application. This offloading would allow SSaaS-backed applications to focus purely on the secret storage and retrieval side of the SSaaS API while dedicated management utilities handle the access control and auditing side of the SSaaS API.

7.3 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 with SSaaS-backed keys model (Chapters 1, 3, and 6). In such applications, local programs handle the encryption and verification of data, storing only encrypted data with third parities and/or on high risk devices while storing the associated encryption keys with one or more SSPs.

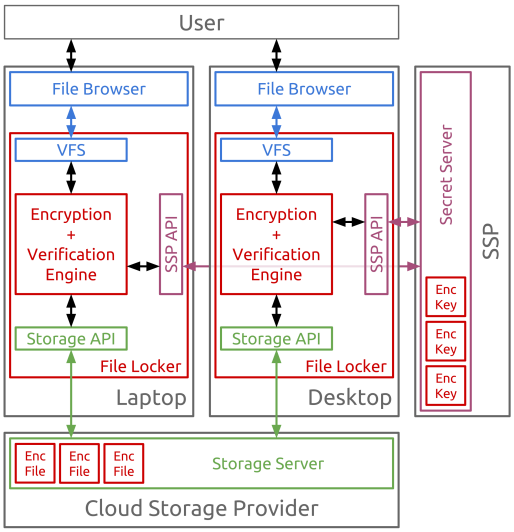


Figure 7.1: SSaaS-Backed Cloud File Locker System

7.3.1 Cloud File Sync/Storage

Building on the original motivating example in this dissertation, i.e. using Dropbox without trusting Dropbox, figure 7.1 illustrates an SSaaS-backed secure cloud storage application. As in the general storage case, this application involves applying client-side encryption/decryption (e.g. using AES [202]) and signing/verification (e.g. using CMAC [71]) 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 the 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, the application stores all required encryption and authentication keys 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 the 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 through the use of SSPs for key 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.

7.3.2 Server Data Encryption

Beyond consumer-oriented encryption systems, there is a 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, an 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 passphrase 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 passphrase to decrypt the system. Since administrators generally lack both physical access to datacenter-based servers as well as an interactive presence on such servers, using traditional encryption systems with most cloud servers remains difficult, if not impossible.

These deficiencies can largely be resolved via SSaaS-backed encryption systems – either full disk or file-system oriented. Using SSaaS, a user can configure each server to store their file encryption and verification keys with an SSP (or SSPs). Each server can then request the keys from the SSP at boot or on access to an encrypted file. Such an application can leverage non-interactive authentication techniques such as the contextual techniques discussed in [133] to achieve autonomous operation (e.g. does the system expect a server to be rebooted at a certain time of day?). In more sensitive situations, the SSP's access control system could even keep the user in the loop by asking the user to provide a passphrase 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. In this model, cloud servers would store all data in an encrypted form, ensuring that a physical search of the underlying storage medium or other interference from the data center host would not be easy. Encrypted data would 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.

SSaaS-backed, server-based encryption efforts 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.

7.3.3 Mobile Device Encryption

The SSaaS model also provides benefits for the protection of local storage. Mobile devices such as phones, tablets, and laptops store huge quantities of personal information. Yet these devices are more prone to theft and loss than 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 users.

Traditional device encryption systems have become prevalent in Android and iOS-base mobile devices [7, 78]. 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 (or 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 passphrase.

Mobile 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 is 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 system could be used to protect individual pieces of data separately, as opposed to 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 would provide more flexible access control semantics on mobile devices than the standard Everything/Nothing access model common today. For example, an SSaaS system could be set up 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 [247], but with cryptographic guarantees).

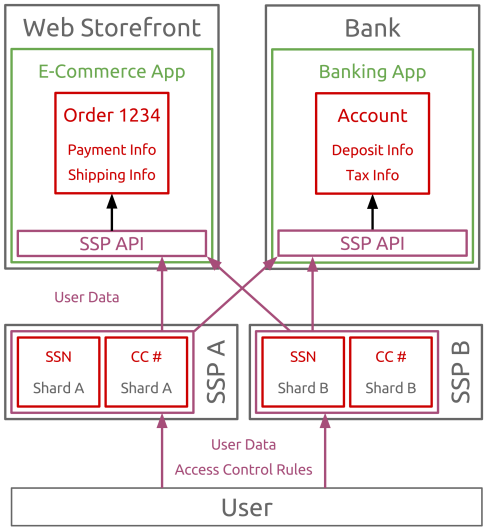


Figure 7.2: SSaaS-Backed Personal Data Repo

7.3.4 Personal Data Repository

Moving past encrypted storage applications, users could also leverage the SSaaS model to create a dedicated repository of personal info (e.g SSN, address, etc). Requiring users to enter personal data on various websites is a common user requirement for placing orders, creating accounts, etc. This practice is undesirable for several reasons. 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 manner that is very difficult for a user to track or control. Instead, an SSaaS-backed data repository would allow a user shard their data across several SSPs and then point websites that require it directly at the SSPs themselves. The user would 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 or changing their name), 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 7.2 illustrates such a use case. In this case, as opposed to the previous applications, the user is storing sensitive user data directly with their

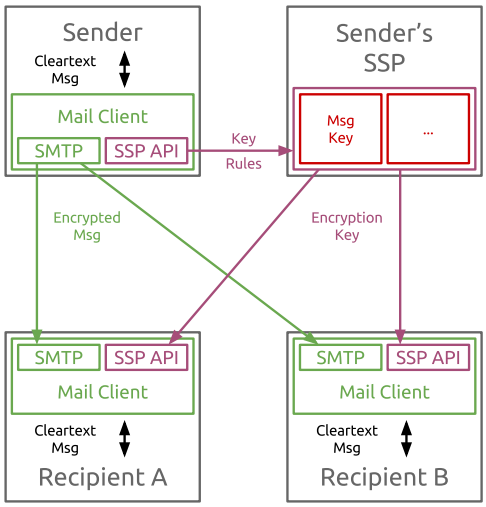


Figure 7.3: SSaaS-Backed Secure Email System

SSPs. As such, it will be important to shard the data across multiple SSPs in order to minimize the trust they must place in each individual SSP.

Such applications would go a long ways toward minimizing the amount of sensitive personal data users 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 users must place in any single SSP and ensures the data is being stored with parties better incentivized to protect it (see the discussion in Section 6.3).

7.4 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 U.S. Government monitoring of email and associated digital communication systems [99, 124, 125]. While solutions like GnuPG [153] allow users to secure the contents of their mail, their complexity tends to limit their use to a small subset of advanced users [122, 345]. The use of SSPs and the SSaaS model could simplify the secure communication paradigm.

Figure 7.3 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 with a locally-generated single-use symmetric encryption key. The client would than 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. 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 fail to support. Such designs have the potential to raise the default level of security inherent in mass digital communication without significantly inconveniencing users or requiring the replacement of existing protocols.

7.5 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 than more commonly deployed shared-secret (i.e. password) authentication systems. Whereas shared secret authentication relies on both the user and the verifier (e.g. server) knowing a common secret such as a passphrase, certificate-based authentication systems rely on a user to be able to cryptographically prove they possesses the private key corresponding to a specific public key. Such systems have a range of applications, from websites to servers and workstations. For example, SSH [350], one of the more common remote-access pro-

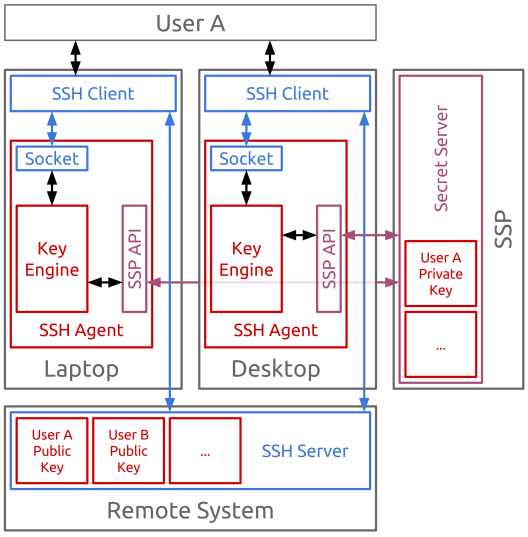


Figure 7.4: SSaaS-Backed SSH Agent

tocols for Unix-like computing systems, uses key-based authentication on both the server (where keys provide server verification) as well as on the client (where keys 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.

7.5.1 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 [54]. Agents are commonly used with popular computing utilities like SSH [350] and GnuPG [153]. 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.

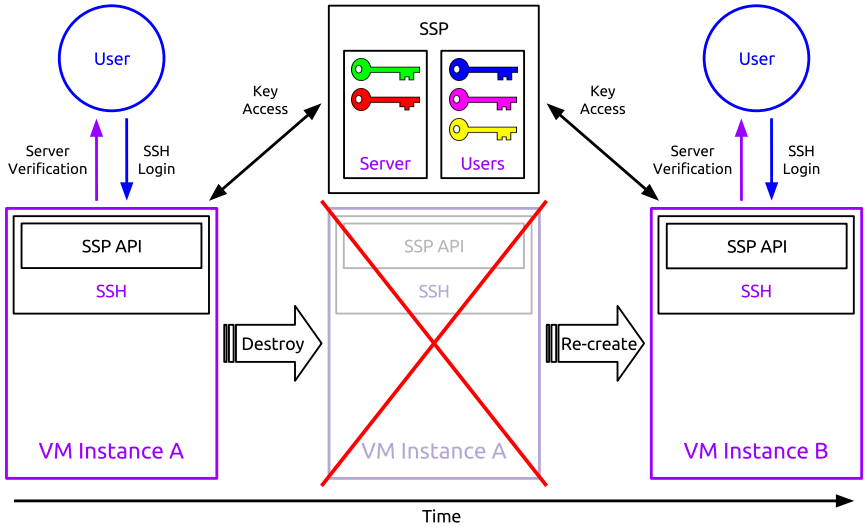


Figure 7.5: SSaaS-Backed SSH Server Key Management

An SSaaS-backed agent can overcome the locality and management challenges associated with traditional cryptographic agent programs. Figure 7.4 shows the basic design for an SSaaS-backed SSH-agent. Instead of storing private keys locally, the 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). 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 [97] and [309]). Such a system also allows large organizations to manage SSH or other cryptographic keys for all their users from a centralized SSaaS management application.

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.

7.5.2 SSH Server Key Management

Similarly, the SSaaS model can provide 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 is important that the user be able to properly authenticate that the server they are accessing is in fact the correct server and not a malicious man-in-the-middle attacker. Unlike SSL/TlS, however, SSH does not rely on a central certificate authority structure for the verification of keys. Instead it uses a "trust on first use" (ToFU) 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 is 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 7.5. 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

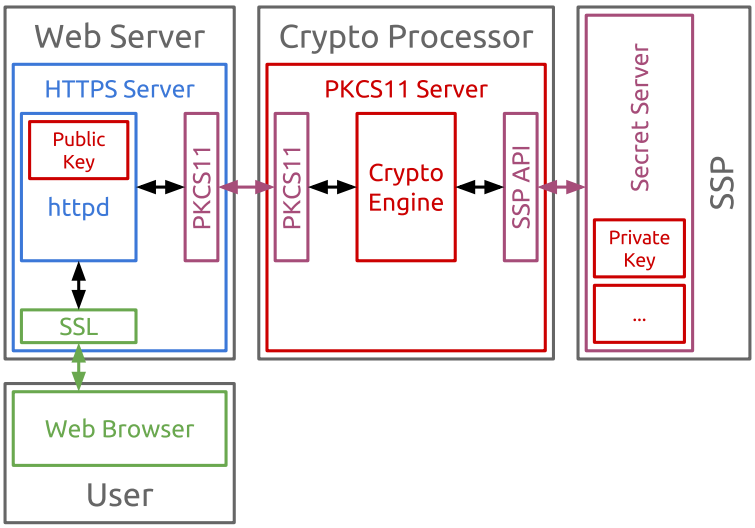


Figure 7.6: SSaaS-Backed Dedicated Crypto-Processor

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 [31, 148]. 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. This solution is the SSaaS equivalent of cloud user management solutions such as JumpCloud [141]. But unlike existing services, an 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 [222], Salt [250], or Puppet [235]. But unlike such solutions, the SSaaS model is designed with the security and privacy of the configuration data it stores as a first principle, not a secondary concern.

7.6 Dedicated Crypto-Processor

The recent Heartbleed bug [48] 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 developers 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 [249]. 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 [177]. Such "softHSM" systems would 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 for an SSaaS-backed system. Figure 7.6 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. Finally, the SSP's auditing functionality ensures that one always

knows 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.

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

Chapter 8

Custos: A First-Generation 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. The chapter provides an overview of Custos and some of its design insights. Additional information is available at [265] and [266].

8.1 Architecture

Figure 8.1 shows the core Custos components. The bulk of Custos's functionality is handled on the SSP server. The Custos SSP server implements the following components:

API: Handles all Custos requests, including requests for ID: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 ID:value objects).

[1] 'Custos' is Latin for "a guard or warden".

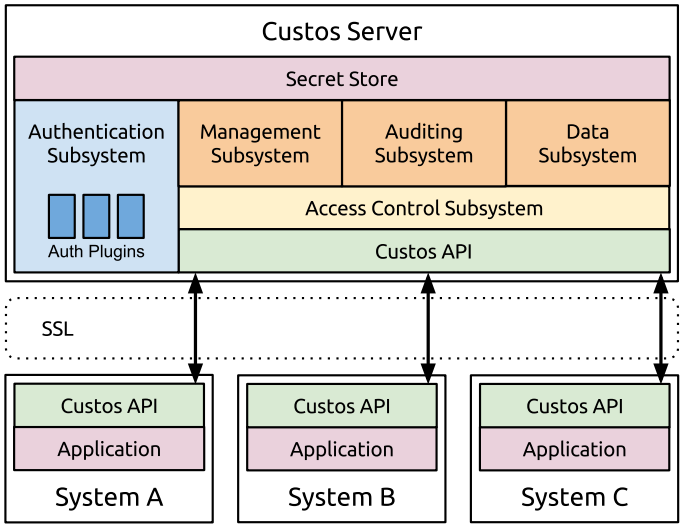


Figure 8.1: Custos's Architecture

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).

8.1.1 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, it is necessary to explain the Custos **organizational units** (OUs): the core Custos data structures. The Custos architecture specifies

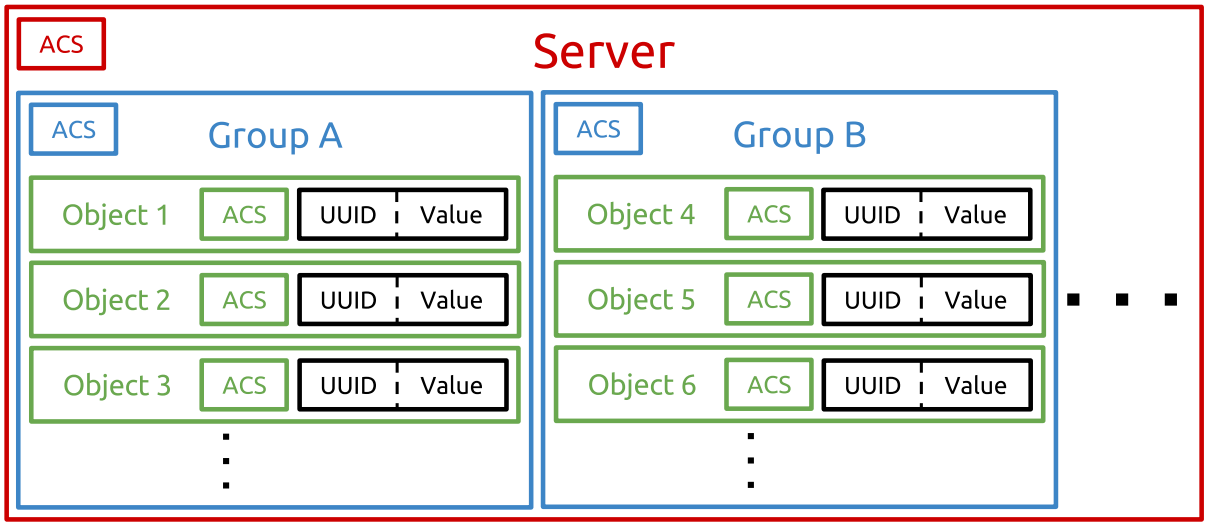


Figure 8.2: Custos's Organizational Units

three organizational units (Figure 8.2): a server, a group, and an ID: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 ID: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 8.3). 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**.

8.1.1.1 Permissions

Each Custos ACS contains a list of permissions: rights to perform specific Custos actions. Custos defines permissions for each OU, e.g. per-server permissions, per-group permissions, and per-object permissions (Table 8.1). Unlike many access control systems, Custos has no notion of

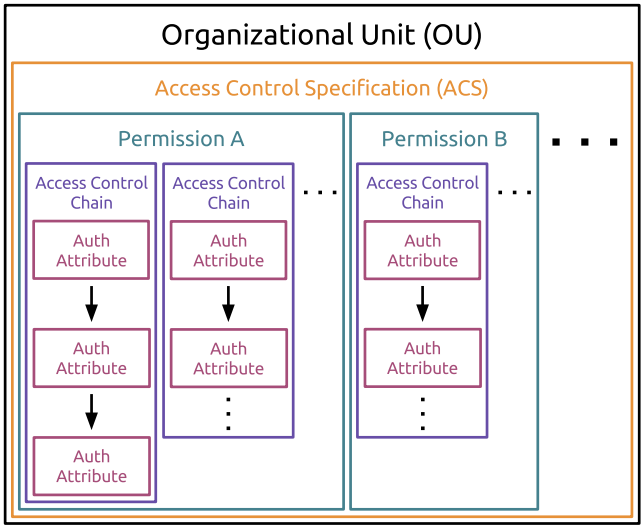


Figure 8.3: Access Control Specification Components

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 an ID:value objects within the given group
grp_obj_list	Group	list ID: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 ID:value object within the given group
obj_read	Object	read the given ID:value object within the given group
obj_update	Object	create a new version of the given ID: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 8.1: Custos Permissions

object 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 includes 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.

8.1.1.2 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 § 8.1.1.3). 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 an ID:value object whose `obj_read` permission has the following ACC set:

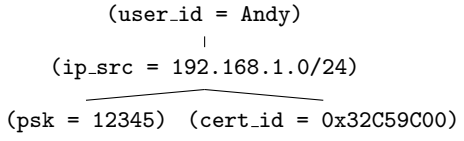
```
[ (user_id = Andy),
  (ip_src = 192.168.1.0/24),
  (psk = 12345) ]
[ (user_id = Andy),
  (ip_src = 192.168.1.0/24),
  (cert_id = 0x32C59C00) ]
[ (user_id = John),
  (psk = Swordfish) ]
```

In order for a read request for the associated ID:value object to succeed, the user would have to make sure that their 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 § 8.1.1.3) and will be automatically appended to our request when received by the Custos server. In order to satisfy it, the user must send their request from the 192.168.1.0 subnet. In the case of the second ACC, the user still needs the 'Andy' username and must satisfy the IP restriction, but this time they must also prove that they have access to the private key associated with the specified authentication certificate instead of providing a password. The third ACC grants access to an additional user, John, with his own password. As long as a user can satisfy at least one ACC in a set of ACCs for a given permission, they 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. An administrator 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.

8.1.1.3 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 8.2: Custos 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 [254]. Examples of potential Custos AA types are shown in Table 8.2.

8.1.2 Protocol

Custos employs a JSON-based RESTful [82] 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 administer a set of secrets and to control who can create new secrets within a group.

The Custos API is secured via TLS/HTTPS [61]. Custos servers are authenticated over TLS via the standard public key infrastructure (PKI) mechanisms (i.e. certificate authorities).[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 [74, 185].

UUIDs [166] 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 [265].

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 the storage and management of a set of secret shares across multiple SSPs is one of the focuses of the Tutamen design discussed in Chapter 9.

8.2 Implementation

Prototype implementations of both a Custos server and a sample Custos client were created to to test and demonstrate the basic SSaaS concepts proposed in this document. These implementation are described below.

8.2.1 SSP Server

The Custos SSP server [257] is implemented in Python. It is designed to use file-system backed key-value stores for 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 [245] web framework to implement the Custos API, and provides HTTPS support via the Apache [16] web server. The server is capable of handling a few hundreds key requests per second, depending on the complexity of the associated ACCs.

8.2.2 EncFS

`EncFS` is a sample SSaaS client [258] created test the Custos SSP server and demonstrate an SSaaS-aware application. `EncFS` implements a FUSE-based [308] layered file system that provides transparent client side-encryption atop an underlying file store. `EncFS` offloads its key storage duties to the Custos SSP Server using the Custos SSaaS protocol. It accomplishes this via `libcustos`, a C-library implementation of the Custos protocol [259].

`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. Nonetheless, 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 [266]. Nonetheless, it is unlikely 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). This latency is also no worse than 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 9

Tutamen: Next-Generation Secret Storage

While Custos (Chapter 8) provides a sound first-generated secret storage system, it has a number of flaws that make solving some of the more nuanced secret storage challenges difficult. Tutamen[1] is a secret storage system design to overcome these challenges. Tutamen applies the lessons learned from building Custos to create a next-generation secret storage system with native support for multi-provider secret sharding and automated secret access.

9.1 A New Approach

Tutamen aims to correct flaws in previous SSaaS systems such a Custos and related first generation secret storage systems such as Vault [129]. In particular, Tutamen aims to provide better support for multi-SSP sharding, for autonomous use cases, and for global-scale scenarios that extended beyond a single administrative domain.

9.1.1 Flaws in Custos

Custos has two main flaws, the correction of which informs the Tutamen design. Namely:

- The Custos access control chain and permissioning system is quite complex. This makes it difficult to properly reason about the net effect of an access control rule or how best to apply that rule in an automated or distributed fashion.

[1] 'Tutamen' is Latin for "a means of protection or defense".

- Custos lacks primitives for easing the synchronization of secret shards across multiple Custos servers. While it is possible to use Custos in a multi-provider manner, the lack of supporting primitives makes doing so more complex than it needs to be.

Tutamen aims to correct these flaws. On the access control front, the true benefit of Custos's access control system is its ability to support out-of-band (i.e. "implicit") authentication attributes (e.g. IP address, etc). Custos's "explicit" attributes (e.g. user-provided passwords, etc) can be better represented via cryptographic authentication practices such as the use of client TLS certificates. Cryptographic methods are both stronger and more convenient than requiring users to provide usernames, passwords, or other shared secrets. Thus, Tutamen replaces all of Custos's in-band (i.e. "explicit") attributes with cryptographically secure TLS client authentication mechanisms. Tutamen then merges Custos's out-of-band ("implicit") attributes into a general purpose (and optional) out-of-band authentication mechanism capable of verifying a range of out-of-band parameters from request attributes such as time and IP address to multi-factor options such as SMS text message or email. Like Custos, this out-of-band system is extensible and plugin-based. Tutamen thus captures the best of the Custos's access control design while also reducing complexity and increasing security.

On the multi-server front, Tutamen adds support for primitives that make it easier to shard Tutamen secrets across multiple servers. Most notably, Tutamen allows clients to request specific secret IDs when storing secrets. This allows a client to reuse a single secret ID across multiple servers and simplifies the mapping of a logically singular secret to multiple shards. Tutamen also separates the secret storage and access control components of the SSaaS model into separate servers, further allowing the user additional flexibility in distributing secret storage duties across multiple providers.

These techniques are discussed in more detail in Section 9.2.

9.1.2 The Ideal Secret Storage System

As discussed in Chapter 6, unlike standard configuration management systems or specific secret storage systems such as password managers, general purpose secret storage presents a number of unique requirements. Such a system must be capable of satisfying the following capabilities:

- Store a wide-range of arbitrary secret data in a secure manner
- Enforce fine-grained access control requirements
- Support a range of authentication sources/methods
- Provide audit logs tracking secret access history

Since the development of Custos, a number of general purpose secret storage systems have been deployed by industry, including HashiCorp's Vault [129], Lyft's Confidant [179], and Square's Keywhiz [290]. These systems exist to fulfill some or all of the requirements listed above. Such systems, however, are hindered by several key limitations. First, they generally require at least one highly trusted server as the basis of their security model, making them unsuitable for operation atop untrusted infrastructure. Second, they tend to lack support for use cases requiring autonomous or remote access to secret material in a secure manner. And finally, they do not scale well beyond a single administrative domain. These deficiencies give rise to two more secret storage requirements:

- Avoidance of the need to place a high degree of trust in any single system outside the application that wishes to store a secret.
- Ability to support a range of secret access use cases, including use cases where automatic or remote access to secrets is required, or where data must be accessible beyond a single administrative domain.

It is toward these final two requirements that Tutamen attempts to advance the state of the art over existing secret storage systems. In particular, Tutamen supports operational modes where no single entity other than the application must be trusted. This allows users to leverage third party secret storage providers running Tutamen servers without having to place high degrees

of trust in any single provider (see Chapter 5). Like Custos, Tutamen also provides support for a modular authentication interface. This interface makes Tutamen suitable for use in situations where it is desirable to leverage external environmental information to automatically evaluate the authenticity of a secret request or where it is necessary to keep a human in the authentication loop without actually requiring that the human be physically present. For example, Tutamen can be used to store the disk encryption keys required to boot a headless server, and only release these keys to the server requesting them when a human responds to a text message confirming the boot request. Finally, Tutamen is designed to support collaboration between loosely-organized users and does not aim to limit its scope to within a single administrative domain.

9.2 Tutamen Architecture

The Tutamen SSaaS platform is designed to handle the storage of arbitrary secret material from a range of applications. To accomplish this, Tutamen employees three discrete architectural components:

Access Control Servers (ACS): The systems responsible for storing and enforcing secret access control requirements and for authenticating secret requests.

Storage Servers (SS): The systems responsible for storing secrets (or shards of secrets).

Applications: The systems leveraging the Tutamen platform to store and retrieve secrets.

The bulk of all Tutamen communication occurs between an application and one or more of each type of server. Inter-server communication is kept to a minimum to support scalability. All communication in Tutamen takes place via TLS/HTTPS [61] connections, and in some cases leverages mutual TLS to provide both client and server authentication. Both access control and storage servers are designed to be used individually or in sets. E.g., an application may store its secrets on a single storage server and delegate access control to a single access control server, or the application may shard its secrets across multiple storage servers and delegate access control to multiple access control servers, or any combination thereof.

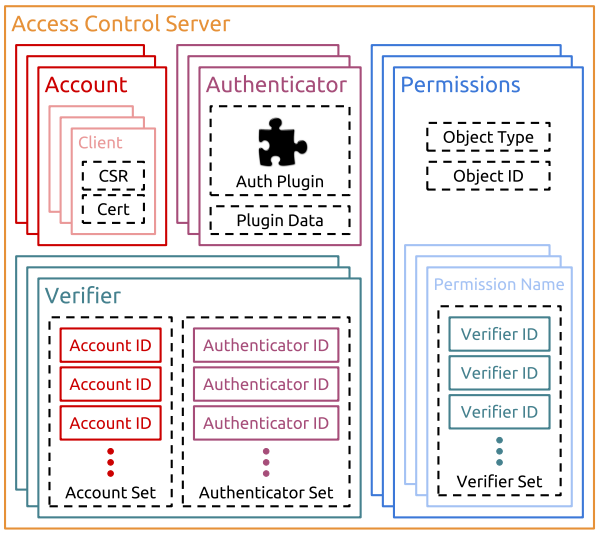


Figure 9.1: Access Control Server Data Structures

9.2.1 Access Control Servers

Tutamen access control servers (ACS) are responsible for authenticating Tutamen requests as well as storing and enforcing all Tutamen access control requirements. Access control servers expose a number of core data structures that reflect the manner in which they operate. Figure 9.1 shows these structures.

In order to track and control access from specific actors, the access control server uses per-actor accounts. These accounts are generally designed to map to individual end users, but they can be used to track any entity to which one wishes to assign specific access control privileges. Accounts thus form the basis of controlling and sharing access to secrets via Tutamen. Within each account are one or more clients. While accounts represent logically singular entities, clients represent specific devices controlled by such entities. Each account has one or more clients. For example, Jane Coworker may have a single account with three clients: one for her laptop, one for her desktop, and one for her phone.

Each client is associated with a single x509 [52] TLS key/cert-pair used to authenticate the client to the access control server. The access control server acts as the Certificate Authority (CA) administering these certificates. When a new client is created it generates a local private key and

uses this key to generate an X509 Certificate Signing Request (CSR). This request is then sent to the access control server where it awaits approval from an existing client in the account. If approved, the CSR is used to generate a signed certificate that is sent back to the new client for use in future ACS communication. To facilitate bootstrapping new accounts, client CSRs are also generated and sent to the AC server during the new account creation process. These CSRs are automatically approved and associated with the new account – i.e., the initial client is created in tandem with a new account while all subsequent clients are approved by previously approved clients.

In addition to accounts, the Tutamen access control server also uses "authenticators". Authenticators are modular mechanisms used to implement contextual access control requirements [133] such as only allowing access during specific times of day or from specific IP addresses. Authenticators can also be used to implement multi-factor and/or out-of-band authentication mechanisms such as confirming approval for a specific request from a user via text message, or otherwise interfacing with external services to gain approval.

Accounts and authenticators are combined via verifiers. A verifier consists of a set of accounts and a set of authenticators. In order to satisfy a verifier, a request must originate from a client associated with one of the member accounts and must satisfy all of the member authenticators. A verifier may contain no authenticators, in which case authorization is granted solely on the basis of accounts.

The final component of the Tutamen access control server is the permissions group. Each permissions group corresponds to a specific object (identified via the combination of an object type and an object ID) within the Tutamen ecosystem. A permissions group contains one or more permissions, each corresponding to a specific class of actions that can be performed on the corresponding object. Each permission is associated with a set of verifiers. In order to be granted a given permission, a request must satisfy at least one of the verifiers in this set.

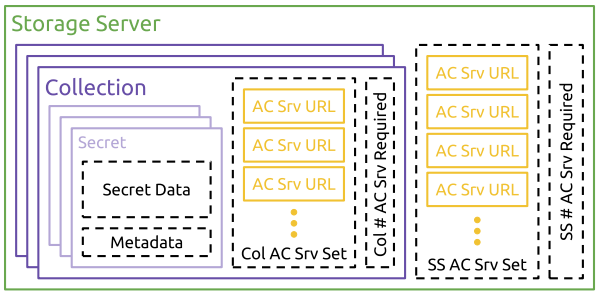


Figure 9.2: Storage Server Data Structures

9.2.2 Storage Servers

Tutamen storage servers (SS) are responsible for storing all or part of each Tutamen secret. Figure 9.2 shows the core storage server data structures.

The top-level data structure employed by storage servers is the "collection". A collection represents a logical grouping of one or more secrets (or parts of secrets). Associated with each collection is a list of one or more access control servers delegated with enforcing the access control requirements for the collection. Access control granularity is thus set at the per-collection, not per-secret level. A collection is also capable of storing user-provided metadata to aid in the mapping of collections to the objects for which they store secrets.

Each collection stores one or more secrets or secret shards. These secrets consist of the actual secret data the applications leveraging Tutamen wishes to store as well as any associated user-provided metadata. Since access control is set at the per-collection level, secrets inherit the access control characteristics of the corresponding collection.

How best to map secret data to collections is left up to each application. This decision is primarily driven by the fact that access control is performed on the per-collection level. Thus, if an application requires that a set of secrets always have a common set of access control requirements (e.g., per-sector encryption keys for an encrypted block device), it becomes efficient to group these secrets into a single collection. Doing so minimizes the complexity of trying to keep access control requirements synced across multiple secrets, and increases performance by minimizing the number

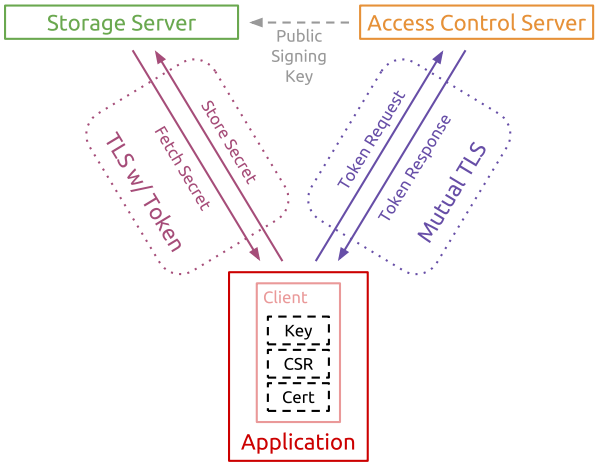


Figure 9.3: Access Control Communication

of requests that the applications must make to secure tokens from the access control server. In cases where each secret requires its own access control requirements (e.g., per-file encryption keys), it is appropriate for the corresponding application to store only a single secret per collection.

9.2.3 Access Control Protocol

Access control servers control access related to both internal (i.e., access control server) and external (i.e., storage server) objects by providing signed authorization tokens in response to valid requests. Similar to previously proposed distributed and federated access control systems [43, 167], each authorization token grants the bearer a specific permission related to a specific object. Unlike previous systems, however, Tutamen is designed to avoid needing to trust any single access control provider (see § 9.2.4). Figure 9.3 shows the basic communication involved in the Tutamen access control process.

Each access control server generates authorization tokens in response to a client sending an authorization request. Each authorization request (and each corresponding token) includes two claims binding it to a specific object: the object type and the object ID. Each token request also contains a claim that binds it to a specific permission (e.g., read, write, delete, modify) for the corresponding object. Authorization requests are further bound to the specific client making the

request (authenticated via mutual-TLS), and to an expiration time after which the token is no longer valid.

Upon receiving an authorization request from a client, the access control server looks up the permission group for the corresponding object (identified via the combination of object type and object ID) and then loads the verifier set corresponding to the requested permission. The server traverses each verifier in this set, verifying both client membership in one of the accounts listed in the verifier as well as executing any authenticator modules required by the verifier until it finds (or fails to find) a verifier that is satisfied by the request. If the server is able to verify compliance with at least one verifier, it grants the authorization request and returns a signed authorization token that includes the object type, object ID, granted permission, and expiration time. The bearer of this token can then present it in conjunction with a request to either the access control server or a storage server in order to be granted the right to perform an approved action on the corresponding object.

Other than the bootstrapping operations and the token request operations themselves, all requests to either storage or access control servers must be accompanied by a valid token. The receiving server validates this token using the public signing key of the associated AC server. For requests to the AC sever itself, this key is available internally. For requests to external storage servers, the signing key is downloaded by the storage server from the access control server and cached for future use. In this manner, access control servers are responsible both for granting and verifying authorization requests and signing the corresponding tokens, as well as for verifying tokens accompanying requests to perform actions on ACS objects (e.g., to create or modify verifiers or accounts). Storage servers are responsible only for verifying tokens accompanying requests to perform actions on SS objects (e.g., to create a collection or read a secret).

9.2.4 Distributed Usage

Tutamen is designed to be used in either centralized or distributed use cases. The simplest Tutamen arrangement (e.g., as shown in Figure 9.3) involves leveraging a single access control

server and a single storage server. In this arrangement, the storage server stores a complete copy of each secret while a sole access control server is charged with enforcing access to these secrets. While this use case is easy to deploy, it has two notable downsides. First, it forces the user to place a high degree of trust in the operator of the access control server (who has complete control over whether or not the access control rules for a given secret are being faithfully enforced), as well as in the operator of the storage server (who, likewise, must faithfully verify incoming tokens and avoid otherwise leaking secret data). Second, it lacks any form of redundancy – if either the access control server or the storage server is unavailable, applications will be unable to retrieve any secrets.

As discussed in Chapter 2, researchers have proposed a variety of systems with the goal of minimizing trust requirements for cloud infrastructure [26, 142, 162, 181, 346]. Tutamen applies similar "minimal-trust" goals to the secret storage problem by offering support for a distributed operation mode as an alternative to single-server operation. Operating Tutamen in a distributed manner is largely a task that is pushed down to the application (or client library). With the exception of offering the necessary primitives to support such operation, both Tutamen storage and access control servers are designed to be largely agnostic as to whether they are being used in a centralized or a distributed manner. This design has the benefit of avoiding server-side scaling challenges, allowing the extra overhead required for distributed operation to be supported by each application that requires it.

Figure 9.4 shows the basic layout of a distributed Tutamen setup. As discussed in Section 5.3, the Tutamen application first shards its secret using a (k, n) threshold scheme [158, 274]. The application chooses the value of n based on the number of storage servers it wishes to utilize. The value of k is then chosen to control how many of the servers must be responsive in order to retrieve the secret; i.e., the difference between n and k controls how much storage redundancy the system has by dictating the number of storage servers that can be unavailable before access to the secret itself is lost. The application then pushes each shard to the n storage servers. If the application is merely concerned about storage redundancy, or about their ability to trust a storage server

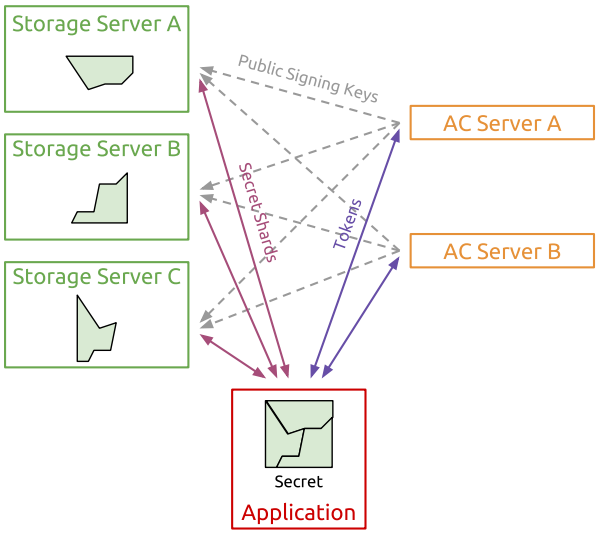


Figure 9.4: Distributed Operation

operator, it can delegate the access control for each secret shard to a single access control server. To retrieve such a secret the application would request the necessary token from the access control server and include it in its request to each storage server for their respective shard of the secret. When the application receives a response from k of the storage servers, it is able to reconstruct the original secret.

In most cases, however, in addition to wishing to mitigate storage server trust and reliability failures, the application will also wish to protect itself against access control server trust and reliability failures. This is accomplished via storage server support for the specification of two pieces of access control metadata corresponding to each stored collection: a list of AC servers approved to provide access control tokens for the collection and a minimum number of servers from which valid tokens must be received. These parameters form the basis of a novel, yet simple (k, n) threshold scheme for access control servers – e.g., a collection may delegate a list of n access control servers from which an application must acquire at least k valid tokens in order to gain access. Thus, if the user does not wish to trust a single access control server, they may require tokens from at least k different AC servers in order to access the data stored in a given collection. As in the

storage case, if the application wishes to withstand the failure of one or more AC servers, it can specify n possible AC servers where $n > k$.

In order to facilitate ease of management when operating in a distributed mode, Tutamen supports allowing applications to request specific UUIDs [166] for each object they create. This allows clients to use the same object ID across multiple servers, alleviating the burden of maintaining a mapping between object IDs and the servers to which they correspond. Using the same object IDs across multiple servers also allows for more efficient token management – e.g., if an application uses the same collection ID on three separate storage serves, all of which delegate a common set of access control servers, it is possible (and desirable) for the application to use a single token granting access to the relevant collection ID on all three servers. Without this capability, an application would be forced to request multiple tokens from each access control server corresponding to the differing collection ID used on each storage server.[2]

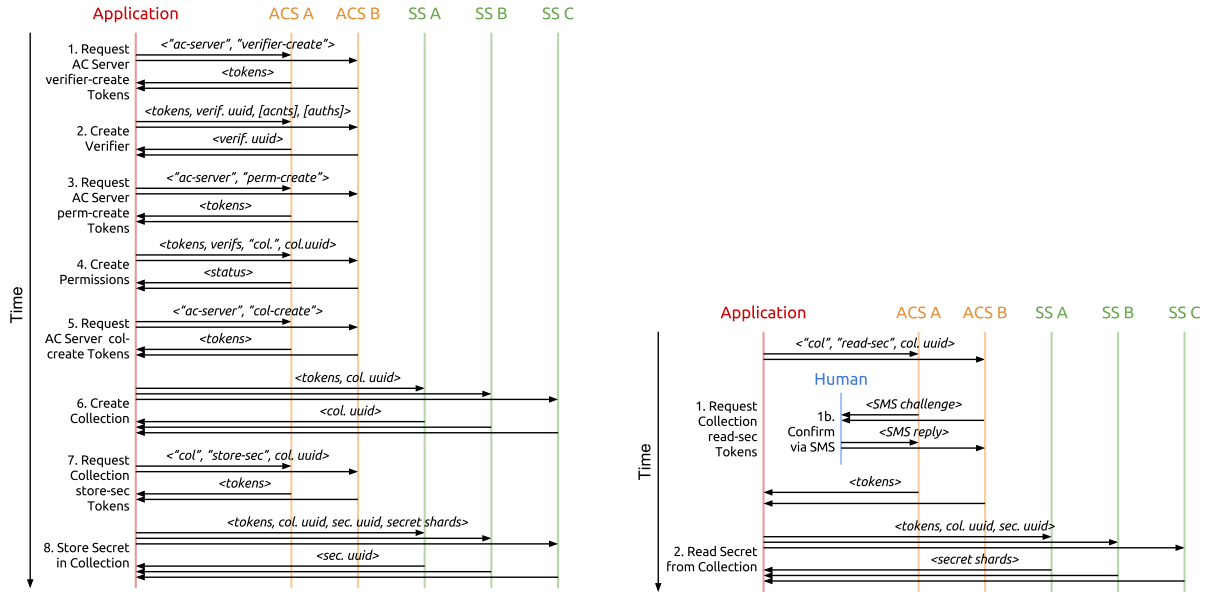
9.2.5 Usage Example

An example of the steps taken by an application to store and then retrieve a secret via Tutamen is useful for illustrating the interaction of the various components of the Tutamen platform. In this example, the Tutamen application is using three storage servers and two access control servers as shown in Figure 9.4. The application has also already bootstrapped an account and client (i.e., it previously contacted the AC servers with a request to create a new account and associated client).

Figure 9.5a shows the steps required to create a new collection and store a secret within it. This diagram assumes the application has already sharded its secret into three parts – one per server.[3]

[2] The ability to request specific object IDs does have one downside: it opens Tutamen up to a possible denial-of-service (DoS) attack where an attacker attempts to request the object IDs they know another application wishes to use for themselves. Since each server may only use each object ID once, the first application to request a given UUID gets it. Thus, if an adversary knew which object IDs a given application planned to use, they could request these object IDs on a set of AC servers for themselves, depriving the original application of the ability to use those servers. Nonetheless, the convenience afforded by allowing applications to request specific object IDs outweighs the potential for DoS abuse. It is also likely at DoS abuse issues can be discouraged via finical incentives - e.g., by charging users for each unique UUID.

[3] Omitted from this diagram is the process of creating verifiers and permissions groups for the collection verifier itself. These permissions groups are necessary to control who can read, modify, or delete the corresponding verifier



(a) Storing a New Secret

(b) Retrieving an Existing Secret (w/ SMS Auth)

Figure 9.5: Tutamen Operations

Figure 9.5b shows the steps required to retrieve an existing secret. This diagram also assumes that the secret in question has an SMS authenticator associated with it, requiring a user to provide a response to an SMS challenge approving access to the secret.

9.3 Tutamen Implementation

Tutamen reference implementations exist for the storage server, access control server, client libraries, and several applications. These implementation allow for the testing of the Tutamen platform and concepts. The details of these implementations are discussed below.

9.3.1 Server Implementation

The Tutamen server implementations expose RESTful interfaces [82] for both the access control and storage server APIs. These interfaces pass data using JSON [55] messages over the

after creation. The process for creating said structures is similar to the process of creating the collection-related verifier and permissions group shown. To avoid the infinite recursion of needing verifiers for each verifier, it is possible for a verifier to be associated with a permissions group in which it is itself a member (i.e., a verifier can enforce its own access control specification).

HTTPS protocol. The full access control server API specification as well as the API reference implementation source code is available at [262]. Likewise, the storage server API specification and source code can be found at [264]. Both implementations are freely available under the terms of the AGPLv3. The prototype servers are written in Python 3 using the Flask web framework [245]. Both servers are designed to be served via WSGI [72] using the Apache HTTP Server [16] for TLS termination and client-certificate verification.

Both Tutamen servers rely on a shared `pytutamen-server` python library for the implementation of their core logic. The `pytutamen-server` source is available at [261] under the terms of the LGPLv3. This library leverages the Redis [241] key-value store for persistent storage. The Tutamen reference implementation adopts the JSON Web Signature (JWS) [139] and JSON Web Token (JWT) [140] specifications for exchanging cryptographically authenticated tokens between Tutamen applications, access control servers, and storage servers. The implementation leverages the pyjwt [225] library for JWS and JWT support. These tokens are attached to subsequent requests using a `tutamen-tokens` header field. The `pytutamen-server` library relies on the Python Cryptography [56] library for CA-related certificate provisioning duties.

The access control servers expose a pluggable authenticator interface through which end users and other developers may add custom authentication functionality. This interface is similar in purpose to previous pluggable authentication interfaces such as PAM [254]. The Tutamen authenticator interface is primarily designed for providing authentication checks beyond the TLS client certificate authentication the access control server automatically performs on every request for the purpose of associating each request with a specific client and account. The reference AC server implementation includes an example SMS authenticator module that allows users to approve Tutamen token requests via SMS text message using the Twilio [315] messaging platform. Additional authenticator modules might include support for only allowing requests during certain times of day or from specific network addresses. Each authenticator plugin is provided with both a set of per-instance configuration data (e.g., to whom an SMS message gets sent for approval) as well as all of the details of a specific token request including both the IDs and metadata associated with

the requesting account and client (e.g., from which information such as originating IP address or time of day can be extracted).

9.3.2 Client Implementation

In addition to the server and authenticator implementations, there are also reference Tutamen client libraries for both Python [260] and Go [197].[4] The Python client library serves as the basis for the reference Tutamen CLI utility [263]. This utility allows users to directly store/retrieve secrets from Tutamen storage servers and to control secret sharing and access control rules on Tutamen AC servers. The CLI is useful for managing Tutamen objects even in cases where other applications (e.g., those discussed in Section 9.3.3) are set up to interface directly with the Tutamen platform. As discussed in Chapter 7, it is not necessary for every SSaaS application to implement all SSaaS functionality. Instead, an application might leverage only the necessary Tutamen commands to perform secret storage and retrieval, leaving the task of managing the sharing of Tutamen-stored secrets to the CLI or to another dedicated management application.

9.3.3 Applications

Tutamen is designed to support a wide range of applications. The reference Tutamen design includes a set of reference applications for the purpose of demonstrating the value derived from using a secure storage system such as Tutamen. These applications all leverage Tutamen's flexibility to achieve functionality that would have been difficult or impossible to achieve without using an SSaaS system like Tutamen.

9.3.3.1 Block Device Encryption

As discussed in Chapters 3 and 7, block device encryption systems are a popular means of protecting the data stored on computing systems in the event that the system is lost, stolen, or otherwise physically compromised. Block-level encryption systems such as dm-crypt [42] (generally

[4] The Go client library was created by Matthew Monaco.

coupled with a Linux Unified Key Setup (LUKS) [91] container) or the QEMU [23] qcow2 encryption system provide methods for securing the data stored on laptops, desktops, and VMs. Such systems traditionally bootstrap security by requiring the user to enter a password at boot-time to unlock a locally stored encryption key which is then used to decrypt the block device in question. Unfortunately, this "human-at-keyboard" security root make such systems difficult or impossible to use atop headless servers or in other situations where no human can be expected to be physically present at boot-time. Interfacing Tutamen with such applications overcomes this barrier.

In order to add Tutamen support to dm-crypt/LUKS, it was necessary to create a Tutamen-aware implementation [196] of the `systemd` Password Agent Specification [307].[5] This specification is used by the LUKS/dm-crypt `cryptsetup` utility to request the necessary decryption secret. At boot-time, `cryptsetup` will send out a request for this secret. Normally this triggers a "human-at-keyboard" prompt for a boot passphrase. A Tutamen-aware password agent can instead respond to such requests by retrieving the necessary decryption secret from a Tutamen storage server (after first retrieving the necessary tokens from the corresponding Tutamen AC server). In order to support Tutamen's use in a pre-boot environment, several modifications had to be made to the `initrd` creation process to add Tutamen networking support, the necessary Tutamen client TLS key pair, and a config file specifying which Tutamen servers to use and the UUIDs of the relevant Tutamen collection and secrets.

In addition to dm-crypt/LUKS support, the Tutamen reference implementation also supports QEMU's `qcow2` VM disk encryption system [267]. Similar to the dm-crypt setup, QEMU normally requires the user to provide the encryption key via the QEMU console when a VM launches. A SSaaS-backed QEMU encryption system replaces this "human-at-keyboard" process with Tutamen-based secret retrieval. In addition, QEMU currently requires the user to provide the full encryption key, not just a passphrase to unlock a pre-stored key [25]. This has negative security repercussions in the common case where users pick short password-like keys. Using Tutamen, users can overcome this barrier since Tutamen servers have no qualms about needing to store or remember sufficiently

[5] Matt Monaco is primarily responsible for the Tutamen dm-crypt integration.

long encryption keys. This system thus increases the security and ease of use of QEMU's `qcow2` encryption.

Using these setups, a user is able to boot servers and VM images with encrypted disks without requiring a human to be physically present at the machine. In cases where a user still desires human approval of the boot process, they can leverage the SMS authenticator module to get an on-demand confirmation from a designated human as a prerequisite to Tutamen releasing the correct key. This allows the user to gain the same level of human-in-the-loop security provided by a typed passphrase, but without actually requiring a human to go to the datacenter to type one in. In situations where one doesn't desire a human-in-the-loop at all, Tutamen-backed systems could support automating the approval process via the use of time-of-day and IP-source authenticators.

9.3.3.2 Encrypted Cloud File Storage

Cloud-based file storage systems such as Dropbox [65] are extremely popular today. Unfortunately, these systems require users to trust the cloud provider with full access to their (generally unencrypted) data. Users wishing to overcome this deficiency can optionally encrypt all of their data on the client before syncing it to the cloud storage provider, but as discussed in Chapter 3, doing so does not generally interact well with such services' sharing and multi-device use cases.

Tutamen provides a solution to this problem by offering a secure key-sharing mechanism. Instead of manually distributing or sharing encryption keys, the user can store their key as a Tutamen secret and leverage Tutamen's access control features to share the secret with the accounts of their friends. This entire process could even be automated such that when a user shares a file via Dropbox, the corresponding encryption key is automatically shared via Tutamen.

Toward this end, the Tutamen reference implementation includes FuseBox: an alternate Dropbox client that performs client-side encryption of all Dropbox files, storing the corresponding encryption keys on the reference Tutamen server [14].[6] FuseBox achieves goals similar to those achieved by [103], but without requiring out-of-band key management. Similar to other file-system-

[6] Taylor Andrews is primarily responsible for the Tutamen FuseBox implementation.

level encryption systems [34, 46, 127], FuseBox provides transparent file encryption to end users. In order to avoid the storage space and security challenges presented by locally caching all Dropbox data (i.e., the operation mode for the official Dropbox client), FuseBox uses AES [57, 202] in a stream cipher mode to transparently stream and encrypt data to/from Dropbox's servers on demand.

Since FuseBox leverages Tutamen to store each per-file encryption key, it becomes possible to share an encrypted file via Dropbox, share its encryption key via Tutamen, and achieve the same level of functionality traditional Dropbox users have without having to expose one's data to Dropbox. While the key sharing process in FuseBox is not yet directly synced with Dropbox's file sharing system, the Tutamen CLI can be used to quickly share the encryption keys between users. In this manner, users can use FuseBox to store and share encrypted files with nearly the same ease with which they might use the traditional unencrypted Dropbox client. By leveraging Tutamen, FuseBox also gains the ability to remotely revoke file access, e.g., in the case a device is lost or stolen, similar to systems such as [97]. While FuseBox does not currently authenticate data, such support could be add to FuseBox's streaming architecture using techniques such as those described at [187]. FuseBox, via Tutamen's distributed operation mode, also avoids the sharing pitfalls associated many existing "secure cloud storage" providers [347] by avoiding reliance on a single trusted party or CA to facilitate sharing operations.

9.3.4 Other

Since the reference Tutamen-backed `ask-password` port [196] speaks the standard `systemd` Password Agent protocol, it can also be used to provide Tutamen-backed passphrase storage to any applications leveraging this protocol. This includes OpenVPN [220] and various password storage utilities. As such, the Tutamen-backed `systemd` password agent is potentially useful in a wide range of situations beyond just full disk encryption. The experience of integrating the `systemd` password agent with Tutamen also suggests that Tutamen would provide a useful backend for a variety of

other "agent" protocols (e.g., [54, 350]), adding support for the kinds of SSH key management applications discussed in Chapter 7.

9.4 Security and Trust in Tutamen

One of Tutamen's primary design goals is its ability to support a wide range of security and trust requirements. It achieves this goal through its support for both centralized and distributed operation as well as through its support for a range of authentication mechanisms.

9.4.1 Security of Individual Servers

The security of each individual access control server rests on several requirements. Failure to uphold these requirements will result in the failure of any security guarantees provided by the AC server.

Certificate Authority Role: Each access control server acts as a CA delegated with issuing and verifying client certificates. Thus, each AC server must store its CA keys in a secure manner and faithfully verify the certificate presented by each client connection.

Token Issuance and Verification: Each access control server is responsible for verifying the access control requirements bound to specific object/permission combinations, issuing signed tokens attesting to such verification, and verifying the signatures of the tokens it receives from clients wishing to operate on access control objects. Thus, each AC server must store its private token signing key in a secure manner and faithfully verify both the access control requirements governing specific token requests as well as the signatures on all incoming tokens.

The storage servers must uphold the following security requirements. Failure to do so results in a failure of the security of the storage server.

Token Verification: Each storage server must securely (via HTTPS) obtain the public token signing key from each AC server delegated with providing access control for a given storage

object. The storage server must then use these keys to faithfully verify the signatures on all tokens it receives. Assuming the token signature is valid, the storage server must faithfully enforce the claims asserted in a given token by only allowing actions granted by the permission contained in the token on the object the token identifies prior to the expiration time specified by the token.

Secure Storage: Each storage server must take steps to store user-provided secrets in a secure manner, releasing them only to requests accompanied by the requisite number of valid tokens granting such release.

Since the tokens the storage server must verify are provided by the AC servers, the security of the storage server with respect to a given collection is dependent on the security of any designated AC servers associated with said collection. If these AC servers are insecure, the objects that delegate access to them will also be insecure.

9.4.2 Security of Multiple Servers

Unlike existing secret management systems [129, 179, 290], the Tutamen architecture is capable of remaining secure even when individual storage or access control servers fail to meet their security requirements. Such failures may result from any of the violations discussed in Chapter 5 including physical server compromise, software bugs, malicious intent, incompetence or legal obligations (e.g., being forced to turn over stored secrets in response to legal or political pressure).

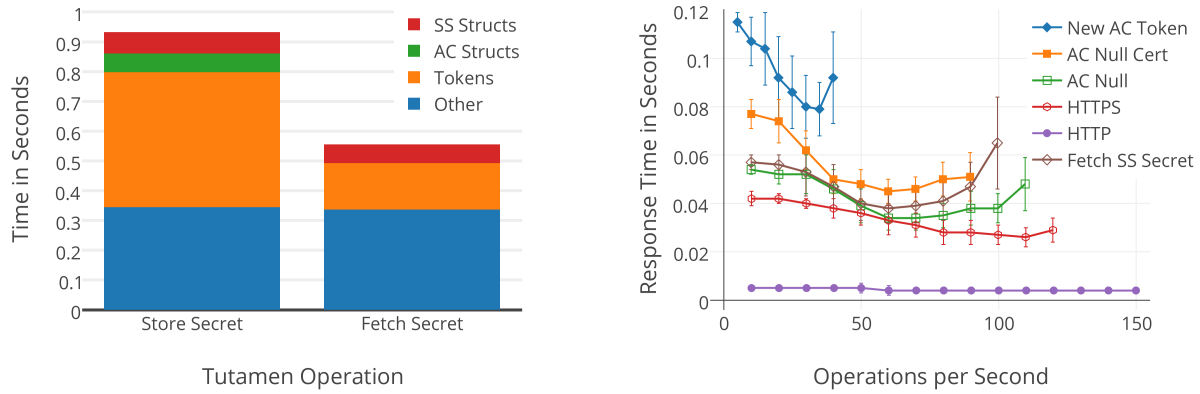
To work around security failures of individual servers, Tutamen applications can leverage Tutamen's distributed operation modes. In these modes, the security of the system as a whole is diffused, no longer relying on the security of any specific access control or storage server in order to keep an application's secrets secure. As described in Section 9.2.4, each application can distribute both secret storage and access control delegations using n choose k schemes. In such setups, the difference between n and k dictates the degree of redundancy inherent in the system while the value of k dictates the degree by which a Tutamen application can withstand security failures. For

example, an application which chooses to shard its secrets across six storage servers where any three shards are sufficient to recreate the secret ($n = 6$, $k = 3$) will continue to remain secure even if two (i.e. $k - 1$) of the storage servers fail to meet their security obligations. The system will also remain available even if three (i.e. $n - k$) of the servers fail to return secret shards. Similarly, if each secret shard delegates six possible AC servers, tokens from three of which are required to grant secret access, the application can withstand the failure of two AC servers to uphold their security guarantees and the failure of three AC servers to return tokens.

9.4.3 Trust Model

Trust in Tutamen follows from the security models of both individual Tutamen servers and of the distributed Tutamen deployment architectures. If a Tutamen application is leveraging only a single storage and AC server, the application is placing a moderate degree of trust in both servers (and by proxy, the operators of both servers). This level of trust may be appropriate for some use cases (e.g., when a user is operating their own Tutamen's servers), but is inappropriate in many other cases (e.g., when using third party hosted Tutamen servers). Fortunately, Tutamen allows applications to avoid placing such a degree of trust in any single server by leveraging multiple servers and picking k and n in a manner commensurate with the degree to which each server is trusted.

Beyond minimizing the amount of trust placed in each individual Tutamen server by leveraging multiple servers, Tutamen also follows the SSaaS pattern of aligning trust with economic incentives (see § 6.2). The Tutamen protocol is standardized and designed to support a range of interchangeable ACS and SS providers. Such a design allows for the development of a Tutamen server marketplace where both ACS and SS providers can compete against each other on the basis of trustworthiness, features (e.g., what types of authenticators they support), and cost. In such an ecosystem, Tutamen service providers who fail to uphold the Tutamen security requirements on their servers will suffer a negative economic effect, disincentivizing such behavior. It is also likely that storage providers who take additional steps to protect the secrets they store (e.g., by using



(a) Timings for Tutamen Operations

(b) Throughput vs Latency Curves

Figure 9.6: Tutamen Performance

systems such as Trusted Platform Modules (TPMs) to encrypt the secrets they hold and harden server security) would be able to command a higher price in the marketplace, incentivizing such best practices.

Thus, unlike other third party cloud services where trustworthiness and economic incentives are in direct competition (as is the case on many "free" third part services that depend on selling user data in order to generate revenue), Tutamen encourages a system where economic incentives are well aligned with user trust. That fact, coupled with the high degree of control over third party trust Tutamen grants by allowing each application to select how many servers to diffuse trust across, make Tutamen a robust system in the face of both security and trustworthiness failures. Such robustness is a critical component of the SSaaS model.

9.5 Tutamen Evaluation

The Tutamen reference implementation has been evaluated in a variety of scenarios using the applications described in § 9.3.3. These scenarios have proven Tutamen's usefulness as an enabler of previously unattainable functionally. While Tutamen is still a prototype, it provides a well-designed architecture capable of supporting a wide range of practical secret storage applications. The use of

strong encryption in these applications would be difficult (or at least burdensome on the end user) to realize without a system such as Tutamen.

Since the Tutamen reference implementation has not yet been optimized for performance, it is more useful to compare relative computational loads than the absolute speed of the Tutamen system. Such a comparison can be achieved by profiling the amount of time the Tutamen CLI utility spends performing various parts of each of these two Tutamen operations. Figure 9.6a shows the time required to complete two of the most common Tutamen operations: storing a new secret and retrieving a previously stored secret. In both operations, the bulk of the server-related run time is spent requesting and retrieving the authorization tokens required to complete the associated operations. In the secret creation case, five tokens are required. In the secret read case, only a single token is required. The remainder of the server-related time is spent either creating AC and storage data structures (as in the store secret case), or reading existing data structures (as in the retrieve secret case). The "other" time is spent reading the Tutamen config files, loading the necessary client certificates, and dealing with the overhead required to interpret the python-based CLI.

It is not unexpected that a Tutamen client must spend the bulk of its time requesting tokens and waiting for them to be approved – token verification is the primary role the access control server must perform, and depending on the complexity of the verifiers associated with the permission the token is requesting, verification can be a fairly complex task. These measurements were obtained using a simple verifier that only required client membership in a specific account. Verifiers that include human-in-the-loop authenticators (e.g., SMS approval) would increase the token turnaround time by the amount of time the human requires to provide approval. Thus, it is important that Tutamen applications treat token approval as an operation that can take anywhere from under a second to human-scale times (e.g., 10s of seconds to multiple minutes). To help alleviate these delays on applications that must perform a high number of Tutamen requests, Tutamen tokens may be reused up until their expiration time. Thus, it is possible for an application to request a

long-lived token and to reuse this token to access multiple secrets within the collection to which the token grants read access.

Figure 9.6b shows the request rate vs response time (with standard deviations) of a token request operation, two "null" AC API operations (one that sends and verifies the client TLS certificate and one that does not), and a fetch secret operation. Raw Apache HTTPS and HTTP curves are also shown for comparison. As these curves show, token verification for the reference server tops out at around 40 requests/second (rps) on a modest server (2-core, 4GB VM running atop 2011-era Intel Xeon hardware). The null AC API operation with client certificates tops out around 90 rps, and the null operation without client certificates tops out at about 110 rps. Raw HTTPS tops out around 120 rps. HTTP topped out around 5000 rps (curve truncated for viewability). The current server setup is thus primarily limited by the cryptography overhead required to serve the application and verify client certificates using TLS. Token verification itself also incurs additional computational requirements, including cryptographic signing operations, but is well within the order of magnitude of the underlying server limits. Secret retrieval (after acquiring a token) is relatively quick, topping out at around 100 rps.

While these levels of performance would not likely meet the requirements of a production-level Tutamen AC server, they have been perfectly adequate for supporting the Tutamen applications discussed in § 9.3.3. Since most of these Tutamen applications require only a single Tutamen secret retrieval at relatively rare rates (e.g., once per server reboot or once per file open), the 40+ requests per second the AC server can provide have been more than adequate for their needs. The Tutamen reference servers are also designed to be horizontally scalable (e.g., by spinning up multiple load-balanced servers that share a common database). That scalability, coupled with future performance-related code optimization, suggests that the Tutamen server infrastructure can be adopted to meet the needs of larger installations with only moderate additional effort.

Chapter 10

Policy Implications

The work presented in this document implicates a number of policy questions in the security and privacy space. A number of these implications have been alluded to in the previous chapters. This chapter explores some of these policy questions in greater depth.

10.1 Toward a Healthy SSaaS Ecosystem

If the Secret Storage as a Service (SSaaS) ecosystem is to flourish, the public must first be convinced that it is a viable solution for increasing the security and privacy of the devices and services they use (and that it does not have a detrimental impact on their ability to utilize such devices and services). While policy initiatives can do little to convince users that the SSaaS ecosystem does not impose an undue burden on their desired use cases,[1] policy initiatives can help to encourage the development of a healthy SSaaS ecosystem. Establishing and maintaining a healthy SSaaS ecosystem relies on three main concepts:

Availability of Cryptography: Users must be able to access and use underlying SSaaS technologies before any SSaaS system can be deployed. An SSaaS ecosystem depends heavily on a range of cryptographic primitives, and these primitives are often the target of regulation. An inability to access or use such primitives will make an SSaaS ecosystem impossible to build.

[1] That is the job of previous chapters in this document.

Maximizing SSP Trustworthiness: In order for the SSaaS ecosystem to flourish, users must believe that, in general, Secret Storage Providers (SSPs) are trustworthy actors. While a user can take steps to minimize how much they must trust each SSP (see next point), if SSPs are viewed as unreliable and untrustworthy in general, the SSaaS ecosystem will be tainted. In addition, users must have faith in the underlying SSaaS protocols and technologies.

Minimizing Mandatory Trust: While a degree of trust in both the SSaaS protocols as well as in each SSP is useful to help the SSaaS ecosystem thrive, it is also important for users to have tools for minimizing the degree to which they must trust any single part of or party in the SSaaS system. Such tools act as the final security and privacy backstop for users, ensuring that their data might remain secure even if their trust in specific protocols or SSPs turns out to be misplaced.

There are a number of policy initiatives that will help to encourage the development of these user views and by proxy, the growth of a healthy SSaaS ecosystem. Such an ecosystem serves the public interest by increasing the security and privacy of users everywhere. These initiatives are discussed throughout the remainder of this chapter.

10.2 Availability of Cryptography

Before an SSaaS system can thrive, it is critical that users have access to the technologies underlying the SSaaS idea. This encompasses a wide range of technologies from networking to cryptography. It is cryptography, however, that tends to incite the most regulatory actions, and so it is the regulation of cryptography that will be the focus of this section.

Many of the ideas proposed in this document rely on the use of a variety of cryptographic primitives, including symmetric encryption, message authentication codes (MAC), asymmetric encryption, asymmetric signatures, etc. Such cryptographic primitives, however, have a checkered

legal and policy history, at least in the United States.[2] This section provides an overview of cryptography-related policy concerns.

10.2.1 A Brief History of Cryptography Regulation

Strong cryptography is a relatively recent development in the timeline of human history. Modern digital cryptography didn't truly come about until the end World War II and the advent of information theory [275]. Prior to that point in time, most "cryptography" systems were based on folk-theory, obscure languages, mechanical devices, or other items not soundly rooted in mathematical theory. Even though the groundwork was laid by the end of World War II, modern practical digital cryptography systems didn't really become available until the standardization of DES in the mid 1970s [201] and the invention of asymmetric cryptography around the same time [62]. The combination of a standardized symmetric key algorithm coupled with the ability to negotiate and/or exchange symmetric keys over insecure channels using asymmetric techniques made digital cryptography a practical tool for securing communications. The security of such systems rests on the provable difficulty of solving certain classes of mathematical problems – a far stronger security guarantee than that provided by earlier systems.

From the late 1970s through the 1980s, digital cryptography remained largely a tool for governments, militaries, and financial corporations. As such, cryptography was often classified as a "munition" and placed under various arms and export control laws regulating the manner in which encryption technology could be distributed outside of the United States.[3] During this time, cryptographically secure systems were not widely used by individuals not associated with the aforementioned institutions.

[2] The bulk of this chapter will focus on U.S. policy and laws since that is the jurisdiction in which the author resides and is most intimately familiar. Similar ideas are applicable to other jurisdictions.

[3] The U.S. did allow the export of certain types of purposely weak cryptography, e.g. cryptography with key lengths below 40 bits. This had the extremely unfortunate effect of encouraging the standardization and proliferation of various weak cryptography algorithms in a wide range of consumer-facing software in order to allow the international distribution of such software. The proliferation of such "export-grade" cryptography still comes back to bite us today – many applications still contain support for such algorithms, and that support makes those products vulnerable to "downgrade" attacks that trick the software into using this weak cryptography. Recent vulnerabilities such as FREAK [28] and Logjam [4] are based on such attacks.

The availability of strong cryptography to the general public begin to increase in the early 1990s with the release of PGP [352]. PGP quickly became the first widely distributed encryption software designed for use by ordinary individuals.[4] PGP's Internet-based distribution also quickly drew the attention of U.S. authorities who viewed such publication as a violation of the export control laws banning the export of cryptographic technologies with key lengths exceeding 40-bits. In defiance of such accusations, PGP's author published the entire source code of PGP as a traditional bound book, the contents of which could be scanned and compiled by anyone with a copy [351]. This publication forced a discussion of the First Amendment [322] issues associated with the distribution of computer source code. Such discussions, coupled with several court cases [327, 328] in favor of the protection of the publication of computer source code under the First Amendment, led to the U.S. government rolling back (although not completely eliminating) most of the export control rules surrounding cryptography by the early 2000s [144].

During this same period, the government tried to standardize a backdoored encryption system known as the Clipper Chip [344]. The chip was designed to be embedded in systems where it would provide "strong" encryption via the Slipjack algorithm while maintaining the government's ability to decrypt such encryption via the use of escrowed master keys. The Clipper Chip quickly fell into disfavor, however, when researchers demonstrated simple methods for bypassing its escrow mechanisms [35]. The publication of such mechanisms was soon followed by discoveries of weaknesses in the Slipjack algorithm [30]. Due to these flaws, and a general distrust of any system requiring the escrowing of encryption keys with the government, the Clipper chip was never widely adopted and the government largely gave up on pushing backdoored encryption systems by the start of the 2000s.

After the failure of both the government's export control based cryptography regulations and their push for key escrow-based encryption standards, efforts to control and regulate cryptography were relaxed through the 2000s and early 2010s. Beyond the government's previous failures, this

[4] Although, as mentioned in Chapter 3, the ability of "ordinary" individuals to actually use PGP is highly questionable [345].

regulatory silence was also due to the fact that the use of encryption such as PGP by end users never really expanded to the levels about which the government seemed most concerned. The usability challenges of encryption technology itself, coupled with the largely apathetic privacy views by many members of the general public, proved far more effective at limiting the widespread use of encryption than any government regulation.

Recently, however, this has begun to change. The Edward Snowden revelations of spying abuses by the U.S. National Security Agency (NSA) in the early 2010s led to a rise in privacy awareness among the general populace [231]. This, in turn, led a push to increase the use of strong cryptography across a wide range of consumer-oriented products, from websites [22] to email [173] to smartphones [78, 7]. The associated increase in end user use of cryptography has led to renewed calls by law enforcement agencies around the world (including the U.S. FBI [49]) to mandate the use of breakable encryption in any scenario where a government feels justified in accessing the underlying data (e.g. when issued a probable cause warrant). Such calls have culminated in the recent court battle between Apple and the FBI over whether or not the FBI can use the All Writs Act [317] to compel Apple to modify the iPhone operating system to expedite the rate at which the FBI can guess the pass-code required to decrypt a phone [15]. Congress has also responded to such calls by suggesting legislation ranging from the formation of encryption "study" committees [337] to outright bans on the use of strong encryption [339] to bans on such bans of encryption [336]. How these kinds of court cases and legislative actions will play out remains to be seen.

10.2.2 A Defense of Cryptography

From the export-control bans of the 1990s [144] to the judicial and Congressional cryptography debates of today, whether or not strong cryptography should be available to the general public is a hotly contested topic. Should the government ever succeed in banning or strongly curtailing the development, distribution, or use of strong cryptography, it will become substantially more difficult (if not impossible) to securely implement SSaaS-related ideas, and by proxy, the privacy and security enhancements SSaaS systems might provide. Indeed, any effort that curtails the use

of or access to strong cryptography is likely to damage the security and privacy of users across a wide range of digital systems.

From a policy perspective, there are a variety arguments against efforts to ban or regulate various form of cryptography. They tend to fall along a three standard tracks:

More Harm Than Good

As discussed in this document and elsewhere [2], cryptography is a critical tool for protecting the security and privacy of modern computing systems. Any effort aimed at weakening cryptography for the purpose of minimizing its effectiveness to "bad" actors is just as likely to minimize its effectiveness to "good" actors. And many more of the "good" rely on cryptography than do the "bad". Efforts to curtail the use of strong cryptography would severely damage the security of financial, communication, and data storage systems. Any backdoor available to the U.S. government is an additional point of failure that an attacker may use to compromise and access a system. Any policy of providing the U.S. government with special access to cryptographically protected data will likely be mirrored by other countries – many of which have far weaker legal protections against the abuse of such powers than does the Unites States.[5] Bans of cryptography trade the security of a large number of normal individuals for the insecurity of a small number of potentially bad actors – and that's not a trade worth making.

Ineffectiveness

Beyond the fact that bans on cryptography are likely to cause more harm than good, such bans are widely believed to be ineffective. Even if the U.S. manages to ban the use of cryptography, cryptographic concepts are no longer a niche topic. There are hundreds of cryptographic products available worldwide, more than half of which are developed and distributed outside of the Untied States [273]. The Internet and modern commerce systems make such products widely available to almost anyone. Thus, even if the U.S. manages to

[5] E.g. if the U.S. government can demand special access, so can China or North Korea – weakening the security of U.S. citizens abroad or anyone wishing to secure themselves and their systems from potentially repressive regimes.

ban cryptography, those wishing to use it will simply obtain cryptography-related products elsewhere. Modern cryptographic concepts are also widely understood and taught [81]. There are a wide variety of individuals capable of creating cryptographically secure systems from scratch.[6] Thus, it would not be difficult for organizations or individuals wishing to employ banned cryptography concepts to find someone to build such systems for them.

Better Options

Law enforcement organizations (LEOs) often cite the "going dark" problem as the reason why cryptographic restrictions are necessary. "Going dark" refers to the idea that as the use of cryptography grows, the ability of law enforcement to monitor criminal activity for the purpose of preventing or prosecuting crimes diminishes [12]. What this viewpoint fails to acknowledge is the fact that this is the golden age of surveillance: law enforcement agencies have access to more data than they ever have had before [303]. Even with the widespread use of encryption, LEOs have more access to data about potential criminals than at any prior point in human history. This fact presents a multitude of ways to combat crime without weakening cryptography. Such mechanisms may require more time and effort than the kind of large scale automated digital surveillance the use of cryptography subverts, but such time and effort is a necessary price to pay to ensure a proper balance between privacy and security. Instead of weakening encryption, law enforcement can focus their efforts on targeted investigations of specific individuals, leveraging the large digital footprints even cryptography-employing users leave behind. And even when using cryptography, individuals are still prone to a wide range of targeted attacks, from efforts to compromise individual systems to social engineering attacks that tend to be very effective against even cryptography employing adversaries. There are numerous tools available to law enforcement to pursue criminals. Weakening encryption need not be one of them.

[6] Although it is not advisable for most individuals to attempt to "roll their own" cryptography if they can avoid needing to do so. Such low-level DIY cryptographic systems have a historically high likelihood of mistakes due to the lack of formal vetting and widespread review.

Beyond policy arguments against regulating or banning cryptography, there are also a number of legal arguments preventing such efforts. In particular, any effort to ban cryptography would raise serious constitutional questions.

10.2.2.1 1st Amendment Issues

Bans on the publication of source code, cryptography-related or otherwise, raise numerous 1st Amendment questions. The free speech guarantees provided by the 1st Amendment have generally been interpreted to apply to source code [327, 328].[7] As such, any ban on cryptography source code is likely an unconstitutional prior restraint on the author's freedom of speech. Traditionally, courts have taken a dim few of laws imposing such prior restraint [294], even in cases where national security may be implicated [297].

Additionally, efforts to mandate the inclusion of a backdoor in certain software are likely a form of unconstitutional compelled speech. Just as blocking the publication of source code interferes with the author's 1st Amendment rights, mandating that an author publishes or includes certain components in their code is also a violation of their free speech rights [299]. While there have been exceptions to such interpretations, it is unlikely the broadness of any general limits of cryptography, coupled with the traditional ineffectiveness of such limits, would meet the kind of "narrowly tailored limits in effective support of a legitimate government interest" requirement such a law would have to overcome to meet constitutional muster.

Still open to debate, however, is whether or not 1st Amendment protections are limited merely to the source code implementing encryption, or whether they apply to executable binaries built from such code as well. If such protections are limited to raw source code, it could effectively limit the widespread use of cryptography to those capable of acquiring and building source code. Such an outcome would have a detrimental impact on the general populace who often lack the ability to perform such actions.[8]

[7] Beyond cryptography, this "code as speech" interpretation is fairly critical to the use of U.S. Copyright Law for the protection of code-related intellectual property.

[8] While the 1st Amendment is likely the best defense of one's right to distribute and use cryptography, one of the more creative constitutional defenses of cryptography [199] comes via the 2nd Amendment's grant to citizens of

10.2.2.2 4th Amendment Issues

Many law enforcement agencies cite the importance of being able to access information for which they have secured a probable cause warrant as their reason for wishing to restrict the use of cryptography. Such opponents fear the ability of cryptography to render such lawfully obtained warrants useless. Whether or not the 4th Amendment [323] can be used to compel a service provider to decrypt data in order to comply with a valid warrant remains to be seen. Traditional interpretations hold that if a third party holds the key necessary to decrypt a user's data (or the data itself), then they can be compelled to provide it to the government – often without requiring a probable cause warrant at all [311]. But what about cases where the third party lacks any ability to decrypt the data? Can they still be forced to aid the government in serving a valid warrant? The answer to this question probably hinges on the "reasonableness" of such a request. Until litigation regarding these issues goes to trial, it is difficult to predict what a court might find.

Cryptography need not be viewed as antithetical to the 4th Amendment. In fact it would seem that cryptography provides the ultimate tool for protecting an individual's 4th Amendment rights since such technology helps ensure that individuals remain "secure in their persons, houses, papers, and effects". But how best to reconcile this protection with a valid "probable cause" warrant remains a matter of debate. And ultimately, as in cases where only the end user possesses the information necessary to decrypt communication, the answer to this question may have more bearing on the 5th Amendment than the 4th. But it seems unlikely that the need to serve a 4th

the right to "keep and bear arms" [324]. The idea is that since the government already has a history of considering cryptography to be a weapon, then the 2nd Amendment should be used to protect the rights of citizens to posses and use cryptography. This theory has never been tested in a court, but it is not entirely unreasonable to believe that cryptography is a necessary component of maintaining a "well regulated militia" today, and thus should be eligible for 2nd Amendment [295] protections. It is also not unreasonable to assume that if the Constitution were being written today, cryptography would seem a much more natural fit for protection via something like the 2nd Amendment than firearms. Whether or not this argument would ever pass judicial scrutiny remains to be seen. And even if it does, the 2nd Amendment only guarantees a weaker set of rights than the 1st Amendment: the 2nd Amendment only protects the right to "keep and bear" cryptography, whereas the 1st Amendment also protects the right to distribute cryptography. It may be possible to take a split approach to protecting cryptography where cryptographic source code and research are protected via the 1st Amendment, while binary programs or hardware employing cryptography are protected via the 2nd Amendment – similar to how the 1st Amendment would protect the publication of a design of a firearm, while the 2nd Amendment would protect the possession of the firearm itself.

Amendment warrant would ever justify the violations of the 1st Amendment that any general ban on the use of cryptography would entail.

10.2.2.3 5th Amendment Issues

Assuming the 1st Amendment can protect an individual's right to distribute, possesses, and use cryptography, then it falls on the 5th Amendment [321] to protect an individual's right not to be forced to disclose the necessary password or key required to decrypt a file or sign a document. The right against self-incrimination has long been held to protect individuals against compelled testimony – which should include being forced to provide a memorized password or key.[9] Courts are currently split on whether or not the 5th Amendment protects individuals from being forced to turn over cryptographic keys or otherwise provide decryption assistance [333, 50, 326].

Further complicating matters is the fact that certain types of cryptography can be designed to decrypt a ciphertext to different plaintexts depending on the key provided. Thus, it is possible that an individual could provide one password to decrypt a file to a picture of a cat and another password to decrypt a file to a plan for robbing a bank. Additional passwords could provide additional alternative contents. In such an arrangement, there is no reliable way to know how many different plaintexts are hidden within a given ciphertext. Even if the government does succeed in compelling an individual to "decrypt" their documents, it is entirely possible they would still wind up with a set of files other than the ones they were seeking. It seems unlikely any court would allow the government to continue compelling a suspect to provide additional keys until they happened to get a decrypted set of files that favored their case, especially since it is possible no such files exist at all.

The 5th Amendment has historically protected the contents of an individual's mind. Cryptography (via encryption) allows an individual to expand the contents of their mind by leveraging a small piece of memorized information (i.e. a password or key) to protect much larger pieces

[9] If the user has written down their password or locked their phone with a fingerprint, 5th Amendment protections no longer apply since the evidence is then physical, not testimonial.

of offloaded information (e.g. a hard drive full of files). Whether or not such an expansion is permissible within the U.S. legal system remains to be decided.[10]

There are clear policy reasons to avoid curtailing or discouraging the use of cryptography, as well as clear legal hurdles to doing so. It is critical that cryptography remains widely and freely available to all, since any reduction in access to cryptography would have severe consequences on a range of security-enhancing and privacy-preserving technologies. This includes on the SSaaS ideas proposed in this document.

10.3 Maximizing SSP Trustworthiness

In order to instill faith in the SSaaS ecosystem, it is important that users view Secret Storage providers (SSPs) as trustworthy entities worthy of protecting user data. At the same time, SSPs are likely to be appealing targets for both legal and extralegal inspection. Whether such services are storing encryption keys for user data or raw user secrets, they will be subjected to a range of government information requests, criminal hacking attempts, and general malfeasance. It is important that SSPs are able to stand up to this barrage of data access attempts and keep the data they store secure. Failure to due so risks soiling the SSaaS "brand". This fact suggests a number of policy-related questions and potential solutions.

10.3.1 Third Party Privacy

SSP servers are most likely to be operated by third parties, and as such, they are likely to be subject to the third party doctrine [311]. This doctrine holds that individuals who voluntarily store their data with third parties have no "reasonable expectation of privacy" [296] for such data. While such a viewpoint may have made sense in the mid 20th century when it was established via

[10] Deciding whether or not such expansions are acceptable using cryptography will likely have a direct implication on a potential future world where human brains are technologically upgradeable. Indeed, there is likely little functional difference between using a cryptographically protected hard drive coupled with a memorized passcode to expand one's "memory" and installing a brain-linked computer chip that expands one's memory. In a world were you have the technology to upgrade the the amount of data your mind can store (and potentially where the government has the technology to "mind-read" by downloading such data), does the 5^{th} Amendment still protect the contents of one's mind? While such questions remain largely academic today, it may not be long before courts are being asked to rule on them.

a series of Supreme Court rulings [298, 300], it does not translate well to a world where storing data with third parties online is the norm.

Before any U.S.-based SSP can be trusted to secure user data, it is necessary for the third party doctrine to be abolished. Users should have a reasonable expectation of privacy for any data they store online, just as they would for any data stored on a hard drive at their home. Fortunately there is movement toward the abolition of the third party doctrine. On the judicial front, the Supreme Court has suggested that it may be time to reevaluate the principle [301]. Congress also seems interested in reevaluating the statute, making progress toward efforts such as reforming the Electronic Communications Privacy Act (ECPA) [332] to include a warrant requirement for digitally stored emails [53].

The abolition of the third party doctrine and the establishment of traditional due process rights for digital data, even when stored with third parties, is a necessary step toward developing a healthy and trustworthy SSaaS ecosystem. Similar protections must be granted worldwide. Data stored via an SSP, regardless of jurisdiction, must be afforded the same degree of respect and legal protection as data stored in a safe at an individual's home.

10.3.2 Privacy Breach Liability

Beyond the need to treat SSP-stored data with the same privacy protections as personally stored data, lies the need to hold SSPs liable for data breaches. If an SSP is breached and user secrets exposed, the user should be able to hold the SSP liable and to seek relief from the SSP for any damage that results. Such liability would follow the growing trend toward holding companies liable for digital data breaches resulting from poor security practices (e.g. [335]).

The nature of this liability could take several forms. The most obvious form would be to impose civil liability commensurate with the value of any exposed secrets on the party responsible for the secure storage of such secrets. This opens up the thorny issue of how to value the loss of a secret. Clearly, some secrets may be worth very little (e.g. an encryption key protecting a set of not particularly sensitive family vacation photos) while others could be worth quite a lot (e.g.

an encryption key protecting a trade secret or other sensitive material) [3]. One way to overcome the valuation challenge would be to have users declare the value of their secrets when they are stored, similar to the manner in which one might declare the value of a parcel when shipping it for the purpose of securing insurance. Given such a declaration, the SSP could charge a user varying amounts: more "valuable" secrets should cost more to store, while less "valuable" secrets cost less. Damages in the event that a secret is lost could then be calculated as a multiple of this value. In cases where the loss is due to an unforeseeable event or otherwise through no fault of the SSP, the user would be reimbursed the declared value of the secret (or potentially some fraction of it). In the case where a secret is lost due to SSP negligence, malpractice, or malfeasance, the user would be reimbursed several multiples of the secret value.

As mentioned in Chapter 6, each SSP would likely be required to secure insurance to cover the cost of such liability payouts in the event that a breach occurs. These insurers, in turn, would charge each SSP an insurance fee on the basis of how "secure" (or the inverse: how "risky") an SSP's storage practice are. Indeed, it is even possible that the government itself might act as such an insurer (or underwriter), as they currently do with banks via the Federal Deposit Insurance Corporation (FDIC) [319]. The need to acquire insurance to cover an SSP's liabilities will also impose a natural limit on the size of any single SSP. Once an SSP's liability grows beyond the size for which it find insurance, it would be forced to stop accepting new customers (or more accurately, new secrets). Such a natural limit on the size and power of any single SSP is a healthy property of a SSaaS ecosystem since it would encourage the competition of a multitude of SSPs and would encourage users to shard their secrets across multiple SSPs to minimize risk. Continuing the FDIC analogy, such a limit is similar to the $250,000 per-account FDIC liability limit which is designed to encourage individuals to spread their money across multiple accounts held with multiple institutions, limiting the trust placed in any single institutions.

The establishment of civil liability for SSPs might also have benefits in the ongoing "forced decryption" and key escrow debates. The very nature of the SSaaS ecosystem likely makes it an appealing model for those wishing to impose key escrow requirements (see Key Escrow discussion

in Chapter 2). Should an SSaaS ecosystem become a standard means of storing secrets such as user encryption keys, it is not inconceivable that the government might wish to pass a requirement that a user escrow a copy of all of their secrets with a government-controlled SSP for use in the event that the government wishes to access a user's encrypted data. As previously discussed, such a practice has a number of flaws and would be bad public policy. Nonetheless, should the government wish to pursue such a policy, having an established system of civil liability for the loss of user secret data would be useful. If such a system were crafted to avoid exempting government-controlled SSPs from civil liability requirements,[11] it could help minimize the number of keys the government chooses to escrow. Indeed, such a system would impose a huge liability risk on the government SSP as a concentrated holder of many secrets. This liability risk attaches a real monetary value to the kinds of existential escrow-related risks raised by security professionals [2]. Making such fears concrete encourages the government to rethink the wisdom of such a system in the first place. If the government can not afford to reimburse the range of companies and individuals who would suffer damage in the event that a government-controlled SSP were breached, they would be unable to operate such an SSP in the first place.

Beyond civil liability, its also possible that it would be appropriate to establish some degree of criminal liability for malpracticing or negligent SSPs. Such liability could mirror existing criminal malpractice and negligence laws such as those imposed on doctors, contractors, and engineers. Such criminal liability would allow the prosecution of particularly bad SSP operators and help to ensure that the SSP market remains relatively safe for consumers wishing to utilize the SSaaS ecosystem.

Regardless of mechanism, establishing a standard system and expectation of secret breach liability will help to incentivize security best practices. Fostering policies encouraging the development of a robust cyber liability security market will further incentivize best practices [291]. Such practices provide users with a measure of relief in the event that their secrets are leaked, increasing adoption of the SSaaS ecosystem.

[11] Although the government does have a habit of exempting itself from such requirements [282], so establishing government liability for SSP-related data loss may be challenging.

10.4 Minimizing Mandatory Trust

Thus far, this chapter has explored mechanisms to ensure entities within the SSaaS ecosystem remain secure and trustworthy. As discussed throughout this document, however, it is also important to provide end users with tools for minimizing the trust they must place in the SSaaS system or its actors.

10.4.1 Protocol Standardization and Transparency

One mechanism for minimizing trust is to ensure that the code and protocols underlying the SSaaS ecosystem (e.g. Custos, Tutamen, etc) remain transparent, open, and unencumbered by IP-related restrictions. These properties will ensure that SSaaS protocols and software can spread and improve over time. In order to achieve these goals, the SSaaS ecosystem must strive for several goals:

Openness of Process: The standardization process itself must be open to the widest possible range of individuals and encourage the participation of such individuals. Furthermore, the process must occur in public so that even those outside of the process may review and comment on the work undertaken within the process.

Availability of Standards: Both drafts and final versions of any standards must be freely and widely available under permissive licensing terms that allow the reproduction and distribution of said standards without requiring any licensing fees.

Unencumbered by Intellectual Property Restrictions: The technology discussed in a standard must be freely available or otherwise unencumbered by the need to secure individual intellectual property licenses or to pay licensing fees. It should be possible for outside parties to develop custom implementation of any standard and for end users to utilize such implementations without either having to secure a license or pay a fee.

This set of requirements helps to ensure that standards can be widely disseminated and adopted, and that a range of competing implementations for these standards can be created and

distributed. Such processes are already used by a variety of standards-setting bodies such as those undertaken by the W3C [348] for the development of web-related standards. Having a similarly open SSaaS standardized process is important to a healthy SSaaS ecosystem in order to encourage the adoption of a single SSaaS standard. Such a standard, in turn, enables users to switch between multiple SSPs and avoids the kinds of SSP lock-in issues that proprietary SSaaS protocols might cause. Having an open and reviewable protocol also reduces the need for users to trust the protocol publisher directly by allowing independent review.

10.4.2 Free and Open Source Implementations

Beyond the importance of developing SSaaS protocols in the open and making them freely available, it is also important that (at least some) implementations of these protocols be open and freely available. Such a requirement is secondary to having a free and open protocol, since a free and open protocol is what allows the development of free and open implementations in the first place. The use of such implementations by SSPs will avoid forcing SSP users to trust that the SSP has faithfully implemented the SSP protocol by facilitating the review of such code by end users themselves (or third parties with an interest in the end user's security).[12]

In order to be truly "open", an SSaaS implementation would need to be both developed in the open and provide users with access to its source code. Such openness can be accomplished by hosting code on a public source code reportedly such as GitHub [101] and by conducting development discussions on an open mailing list. These forms of openness ensure that users are free to review the implementation of the SSaaS protocols themselves, reducing their need to trust the software itself.

[12] Although this leaves open the problem of how to ensure that an SSP is running the open source software they claim to run, as opposed to running software that is actually a malicious or flawed clone. This problem might be dealt with via the insurance and liability issues previously discussed – i.e. insurers might require audits of an SSP's deployed code base and/or make their insurance programs contingent on the use of certain open source SSaaS protocol implementations. Alternatively, it is possible that research from the "reproducible builds" space might assist in this task [58]. How to verify such reproducible builds for third party hosted software remains an open problem.

Beyond openness, it is also important that the software be Free.[13] In the open source community, such freedom is generally premised on four core "rights": the right of the user to run the software as they wish; the right of the user to review and modify the software; the right of the user to redistribute unmodified copies of the software; and the right of the user to redistribute modified copies of the software [89]. Cloud-based software such as that employed by SSPs also benefits from a fifth right: the right of users of the software to obtain a copy of the source code powering the software itself [88]. Granting these rights to users helps to ensure that they are free not only to independently review software, but also to improve and redistribute a piece of software in the event that they dislike what they find while reviewing it. These are important properties since they further allows users to reduce their trust and reliance on any single implementation of the SSaaS protocols.

While having a single standardized protocol is critical for the development of a healthy SSaaS ecosystem, the opposite is true for implementations of this standard. A healthy SSaaS ecosystem should support a variety of implementations (free/open or proprietary) of the SSaaS protocol standard. Such a diversity of implementations allows users to avoid having to trust any single implementation, and decreases the insecurity that can result from a software monoculture. Ensuring there at least a few core SSaaS implementation strains remain free and open will help to encourage the development of additional SSaaS strains, creating this desirable level of diversity.

10.4.3 Use of Multiple SSPs

While openness and transparency provide tools for reducing the trust that users must place in SSaaS software and the ecosystem in general, reducing trust in individual SSPs is best accomplished by sharding user data across multiple SSPs. As discussed in Chapter 5, such sharding allows the user to distribute their secrets in a manner such that no single SSP can derive any meaningful information from the shard of the secret they hold. Since sharding data across SSPs reduces the risk each individual SSP can pose to a user, it is desirable to the health of the SSaaS ecosystem to

[13] "Free" as in freedom, not "free" as in beer.

have policies which encourage such sharding. To accomplish this goal, it is useful to consider the concept of secret sharding from three main angles: limits regulators might impose on the technology underlying and act of sharding; how SSP liability and insurance relates to sharded secrets; and the jurisdictional issues that sharding a secret across multiple SSPs might raise.

10.4.3.1 Limits on Data Sharding

Similar to the previously discussed issues associated with limiting the use of cryptography, any effort to limit or restrict a user's ability to shard their secrets across multiple SSPs must be avoided. There a number of reasons regulators might attempt to limit such sharding. Foremost among these reasons are that the algorithms underlying data sharding (e.g. Shamir's [274]) might be interpreted as a form of cryptography, with all the regulatory baggage that may entail, and that sharding data across multiple SSPs might violate various data localization laws.

On the first point, whether or not secret sharding schemes would be classified as "cryptography" for the purpose of cryptography-related regulations is an open question. Existing U.S. export control rules governing cryptography do not contain a specific definition of what constitutes a "cryptographic" system [329]. A traditional view of cryptography as a means for scrambling information via a key doesn't necessarily apply to secret sharding systems since such systems include no key and aren't really "scrambling" information so much as selectively splitting it into multiple partial copies with special properties.[14] It is possible that efforts to limit the use of cryptography in general might not actually apply to secret sharing schemes. Nonetheless, any effort to limit the use of secret sharding schemes, just like any effort to limit the use of other forms of "cryptography", will prove detrimental to the SSaaS ecosystem. As stated previously, ideally regulators will avoid trying to limit the use of cryptography at all. Short of that, avoiding the classification of secret sharding schemes as cryptography for the purpose of cryptography-related regulations is a desirable

[14] As discussed in Chapter 2, secret sharing schemes are in a distinct class of algorithms from traditional cryptography schemes. Whereas traditional cryptography derives its strength from assumptions about the amount of computing power an adversary can bring to bear solving certain kinds of math problems (conditional security), secret sharing systems derive their security from fundamental properties of information theory, and remain secure regardless of the amount of computational power an adversary possesses (unconditional security).

outcome since it avoids sweeping such schemes into any limits imposed on more traditional forms of cryptography.

On the second point, it is likely that laws aimed at mandating the storage of data in a local jurisdiction (e.g. [60]) will harm a user's ability to effectively shard their secrets across multiple providers. Such laws are generally presented as privacy-enhancing measures aimed at ensuring third parties that store user data are subject to the laws of the jurisdiction in which the user resides. In reality, however, data localization laws are often used for less pure purposes such as maintaining government-access capabilities to user data. This is bad for the SSaaS ecosystem for two reasons. First, any reduction in the number of SSPs available for a user to chose from weakens the competition-driven incentives of the SSaaS ecosystem. Limiting a user to only storing data with local SSPs, or forcing SSPs to store data for a user in a specific jurisdiction, limits the number of SSPs to which a user has access. Access to fewer SSPs means a smaller market with less competition and thus a higher likelihood of poorly or maliciously operated SSPs. Second, protecting user secrets from overly intrusive governments is one of the roles of the SSaaS model. Thus, any effort to limit a user's access to SSPs outside of the jurisdiction in which they reside will subvert one of the potential SSaaS benefits. This idea is discussed further in Section 10.4.3.3.

In all cases, regulators should avoid policies that would decrease a user's access to secret sharding technology or that would limit the number or diversity of secret storage providers to which a user has access. Avoiding both regulations aimed at suppressing access to secret sharing schemes, and regulations encouraging the "Balkanisation" of the Internet into locally-controlled sub-domains [24] is necessary for encouraging a healthy SSaaS ecosystem.

10.4.3.2 Liability and Insurance

Sharding secrets across multiple SSPs also has repercussions for the liability and insurance ideas discussed in Section 10.3.2. The main question secret sharding poses to SSP-related liability ideas is that of the value of a single secret shard. That is, if an SSP is liable for leaks of user secrets, how do we value the harm caused by leaks of a single shard of a secret?

Values for secret shards could range from being equal to the value of the unsharded secret to being worth nothing at all. Making secret shards worth as much as the secret itself makes sense if you assume that the exposure of any secret shard is an unacceptable risk since it increases the risk of the exposure of the entire secret. On the other extreme, valuing secret shards as worthless makes sense if you assume that each shard taken by itself provides no information about the underlying secret, and is thus "worthless" even if leaked. It is likely that the best solution lies somewhere in between. Valuing secrets shards as equal to the value of an unsharded secret will likely make the sharding of secrets cost prohibitive. Assuming that the declared value of a secret is the driving mechanism an SSP will use to charge users for secret storage, then valuing secret shards the same as secrets increases the user's cost to store the secret on the order of a factor of n, where n is the number of SSPs a user wishes to shard their secret across. This represents an undesirable increase in cost. Similarly, valuing secret shards as worthless ignores the realty that the leak of multiple shards can still reveal the secret – imposing the same harm on the user that the leak of an unsharded secret would. But if shards are "worthless", than the user will have no ability to seek restitution for the harms they suffer from the SSPs responsible for leaking each shard. This is also undesirable.

Intuitively, it would make the most sense for the act of secret sharding to have no effective impact on either a user's cost to store the secret nor their ability to recoup losses if the secret shards are leaked. To accomplish such an equivalency, it is necessary to value each shard of an n choose k secret split as being worth v/k where v represents the cost of the unsharded secret and k represents the number of shards necessary to reconstitute the secret. In such a situation, the user's secret is effectively leaked when k shards are exposed. The exposure of k shards allows the user to seek damages for a value equal to v/k from each of the k SSPs responsible, resulting in the same value v the user would be due in the unsharded case. Similarly, the cost to the user to store each secret shard should be equal to c/n where c is the cost to store the unsharded secret and n is the number of total SSPs in use. As in the liability valuation, such a split will ensure the user pays the same for a sharded secret as an unsharded secret, at least in cases where $n = k$. Since n

choose k secret sharding schemes allow for the user to specify an n greater than k for the purpose of adding redundancy to the system, a user may still end up paying more to store a sharded secret than an unsharded secret in any case where $n > k$. In such situations, however, the user is gaining an additional degree of redundancy that they did not have before, justifying the additional cost. Thus, valuing secret shards at v/k for the purpose of liability and charging for their storage at a rate of c/k will avoid financial or liability related disincentives to sharding secrets across multiple providers.

The user can do even better than this in situations where the cost to store a secret is exponentially or geometrically, instead of linearly, related to the declared value of the secret. For example, it is not inconceivable that the cost to store a secret valued at $1000 should be more than 10x the cost to store a secret valued at $10. Such a nonlinear cost vs value curve can be justified by the fact that more valuable secrets are likely to be more desirable targets for attackers, and thus command a higher premium to store relative to their value. Assuming SSPs adopt such non-linear cost regimens, sharding a secret becomes financially beneficial to the user since it minimizes the declared value of each individual shard, reducing the total cost. Whereas a linear cost of secret storage based on secret value results in the same total cost to the user when sharding their secrets across multiple providers, a non-linear secret storage cost can cause the user to realize a savings, further encouraging the practice. It is potentially desirable for SSPs to adopt non-linear secret storage cost regimes where the cost to store a secret increases geometrically or exponentially with its declared value. Such an ecosystem would further encourage users to shard their secrets across multiple providers, increasing user security while reducing both SSP trust and cost.

10.4.3.3 Jurisdictional Issues

The final policy component implicit to the sharding of secrets across multiple SSPs is that of the multi-jurisdictional issues such sharding might raise. If a user limits their sharding of secrets to SSPs located within a single regulatory domain, jurisdictional issues are moot. But it is possible (and, as mentioned previously, desirable) for a user to shard their secret across a range of multi-

jurisdictional SSPs. Such sharding both helps to insulate the user against privacy failures implicit to any single jurisdiction (e.g. spying by oppressive governments) and ensures the user has access to the widest possible diversity of SSPs, increasing the likelihood that market forces can help maximize SSP trustworthiness.

How regulators might view the sharding of secrets across multiple domains is debatable. On one hand, such sharding is likely to subvert legitimate efforts by officials to access stored user data (for example, access granted via a probable cause warrant) due to the need to enforce such access requests across multiple jurisdictions. This is likely to frustrate regulators. On the other hand, multi-national secret sharding has clear benefits for end user security and the availability of user data by minimizing the kind of privacy failure risks that tend to be jurisdiction specific (e.g. the overthrow of a legitimate government by an illegitimate one). Such challenges are not necessarily unique to multi-jurisdictional secret sharding. Activities such as Internet-based gambling have long leveraged conflicts between varying jurisdictional laws to operate in a legal gray area [192]. Additional jurisdictional arbitrage examples range from seasteading [21] to corporate law [152].

Law enforcement organizations (LEOs) wishing to legally reconstitute a multinationally sharded secret face an uphill battle. First, such actors would have to employ one of the variety of mechanisms designed to aid law enforcement in recovering evidence across national borders. Such mechanism often consists of mutual legal assistance treaties (MLATs) laying out a framework for how LEOs in one country can request the assistance of LEOs in another country [292, 342]. Such efforts, however, tend to be slow and burdensome. Multiply such effort by the number of shards a user might have, each stored with an SSP in a different country, and it becomes increasingly difficult to legally obtain the necessary shards to reconstitute a secret [147]. Furthermore, there are some countries that may lack MLATs all together, or that lack the kind of centralized government and rule of law necessary to make an MLAT enforceable. Sharding secrets across multiple multi-national SSPs imposes a significant burden upon anyone wishing to compel such SSPs to provide their shards of the secret.

While this arrangement may appear as a bug to those wishing to enforce lawful data requests, it is probably a feature to those wishing to avoid such requests. Indeed, and as mentioned in Chapter 5, it is likely that certain SSPs would specifically market their resistance to compelled legal orders as a reason why users should select them as an SSP. In fact, SSPs might even encourage a form of international regulatory competition where countries strive to pass more stringent privacy protections in order to attract SSPs and the business they might generate. This outcome is likely good for users on the whole since competition between governments is likely to lead to better privacy-related laws for all users. And given the range of legal abuses practiced by governments around the world, the ability to leverage a multinational network of SSPs to gain more favorable privacy protections is a desirable capability.

But regardless of benefit, it is likely that multinational SSP sharding will be used to subvert certain wholly legitimate investigations. One way to counter this might be pursue efforts to streamline and expedite existing MLAT processes [209]. Indeed, given the automated nature of the SSP ecosystem, it is likely that certain MLAT-related functions could be automated for the benefit of those placing lawful data requests. But even with such efforts, it is unlikely that all SSPs will ever be subject to a single set of legal obligations and requirements, ensuring that some degree of jurisdictional arbitrage always remains possible. These issues are not unique to the SSaaS ecosystem. They effect a wide range of digital services and likely demand a more general solution than the SSaaS ecosystem itself can provide. It is useful to provide LEOs with satisfactory methods for securing lawful access to user data across borders, lest they become discouraged and instead seek to implement detrimental data localization laws that ban the storage of data aboard all together [234]. But as in the cryptography discussion, any effort to grant lawful access to sharded data that reduces a user's ability to shard their data in the first place is likely to cause more harm than good. For every criminal that sharding protects, there will be a multitude of everyday users, dissidents, and other good faith actors that the inability to leverage sharding would harm.

Multinational SSP sharding thus represents yet another level of trust mitigation. Sharding secrets between SSPs allows the user to reduce trust in any single SSP. Sharding secrets between

nations allows the user to reduce trust in any single nation. And given many nations' penchant for privacy abuses, that is a useful trust mitigation to have.

Chapter 11

Conclusion

The magnitude of user data generated and stored on computing devices and with third party providers increases ever day. So too increases the range of threats to which such data is subjected. Building systems that improve the privacy and security of such user data is therefore of critical importance.

Unfortunately, many traditional privacy and security enhancing systems are not well adapted to modern usage practices, leading users to forego the use of such systems all together. Privacy and security enhancing systems must not merely protect user data, they must do so in a manner that preserves a user's ability to use the devices and services they wish in the manner to which they are accustomed. Failure to do so leads to the creation of systems that go underutilized or unused.

One of the major hurdles to building privacy and security enhancing technologies lies in the challenges associated with securing the secrets needed to bootstrap the secure operation of such technologies. Such secrets range from the encryption keys necessary to encrypt user data to the passwords necessary to access online services. Coupled with the need to support desirable use cases, this "secret storage challenge" makes it difficult to build effective security and privacy enhancing systems.

The Secret Storage as a Service (SSaaS) model provides a mechanism for securely storing user secrets in a manner that supports a wide range of use cases. The SSaaS system also provides mechanisms for reducing the trust users must place in third parties, making such trust a severable and tradable commodity in a market favoring increased trustworthiness. The SSaaS model maxi-

mizes a user's ability to flexibly control the manner in which their secrets are used (and by proxy, the manner in which any data such secrets protect is used).

This dissertation makes several key contributions to the state of the art. These include:

- A model for and analysis of third party trust (Chapter 5).
- An overview of the SSaaS model and the privacy and security enhancing benefits it can provide (Chapter 6).
- A discussion of how the SSaaS model can be integrated to improve the security and privacy of a range of user- and developer-facing applications (Chapter 7).
- Custos – a first generation SSaaS prototype (Chapter 8).
- Tutamen – a next generation SSaaS prototype with robust support for sharding secrets across multiple storage providers and autonomous operation (Chapter 9).
- A discussion of the policy implications necessary to build and support a healthy SSaaS ecosystem (Chapter 10).

While this work represents a notable exploration of the SSaaS idea, it is neither exhaustive nor complete. Future work related to this topic might include the study of how the auditing components for the SSaaS model can be leveraged to automate various components of SSaaS security. The Tutamen work presented herein would also benefit from further development to increase its performance capabilities and make it a fully production-ready reference platform. Finally, there is much additional policy work to be done, from efforts aimed at standardizing and encouraging the adoption of an SSaaS protocol to ongoing defenses of cryptography and other core security-enhancing technologies.

The SSaaS model can improve the security and privacy of computing systems across a range of use cases. These improvements will help to ensure that the computing services that have provided so many benefits over the previous 20 years continue to provide benefits for the next 20 years. Flexible secret storage systems such as those proposed herein are a critical component of future secure computing systems.

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] Harold Abelson, Ross Anderson, Steven M. Bellovin, Josh Benaloh, Matt Blaze, Whitfield Diffie, John Gilmore, Matthew Green, Susan Landau, Peter G. Neumann, Ronald L. Rivest, Jeffrey I. Schiller, Bruce Schneier, Michael A. Specter, and Daniel J Weitzner. Keys under doormats: Mandating insecurity by requiring government access to all data and communications. Journal of Cybersecurity, pages 69–79, November 2015.

[3] Alessandro Acquisti, Leslie K John, and George Loewenstein. What is privacy worth? The Journal of Legal Studies, 42(2):249–274, June 2013.

[4] David Adrian, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, Paul Zimmermann, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, and Emmanuel Thomé. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 5–17, New York, New York, USA, 2015. ACM Press.

[5] AgileBits. 1Password. `http://agilebits.com/onepassword`.

[6] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing, STOC '96, pages 99–108, New York, NY, USA, 1996. ACM.

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

[8] Amazon.com, Inc. Amazon information request report. `http://d0.awsstatic.com/certifications/Information_Request_Report.pdf`.

[9] Amazon.com, Inc. Cloud HSM. `http://aws.amazon.com/cloudhsm`.

[10] Amazon.com, Inc. Elastic Cloud Compute. `http://aws.amazon.com/ec2`.

[11] Amazon.com, Inc. Simple Storage Service. `http://aws.amazon.com/s3`.

[12] Nate Anderson. The Internet Police. Norton & Company, 2013.

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

[14] Taylor Andrews. FuseBox source code. `https://github.com/taylorjandrews/FuseBox`.

[15] Sebastion Anthony. Tim Cook says Apple will fight US gov't over court-ordered iPhone backdoor. Ars Technica, February 2016.

[16] Apache Software Foundation. Apache HTTP server project. `https://httpd.apache.org/`.

[17] Apple, Inc. iCloud. `https://www.apple.com/icloud`.

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

[19] Jeff Atwood. Building a computer the Google way. Coding Horror: Programming and Human Factors, March 2007.

[20] Chris Ballinger. ChatSecure: Encrypted messenger for iOS and Android. `http://chatsecure.org`.

[21] O. Shane Balloun. The true obstacle to the autonomy of seasteads: American law enforcement jurisdiction over homesteads on the high seas. University of San Francisco Maritime Law Journal, 24(409), 2012.

[22] Richard Barnes. Deprecating Non-Secure HTTP. Mozilla Security Blog, April 2015.

[23] Fabrice Bellard and Others. QEMU: Open source process emulator. `http://wiki.qemu.org/Main_Page`.

[24] Tim Berners-Lee. Tim Berners-Lee on the web at 25: the past, present and future. Wired, March 2014.

[25] Daniel Berrange. QEMU QCOW2 built-in encryption: Just say no. `https://www.berrange.com/posts/2015/03/17/`, March 2015.

[26] 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.

[27] 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.

[28] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In 2015 IEEE Symposium on Security and Privacy, volume 2015-July, pages 535–552. IEEE, May 2015.

[29] 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.

[30] Eli Biham, Alex Biryukov, Orr Dunkelman, Eran Richardson, and Adi Shamir. Initial observations on Skipjack: Cryptanalysis of Skipjack-3XOR. Selected Areas in Cryptography, (August):362–375, 1998.

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

[32] BitTorrent. Sync. `http://www.getsync.com`.

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

[34] Matt Blaze. A cryptographic file system for UNIX. Proceedings of the First ACM conference on Computer and Communications Security, pages 9–16, 1993.

[35] Matt Blaze. Protocol failure in the escrowed encryption standard. Proceedings of Second ACM Conference on Computer and Communications Security, pages 59–67, April 1994.

[36] Matt Blaze. Oblivious key escrow. Information Hiding, 1174, 1996.

[37] Joseph Bonneau. A technical perspective on the Apple iPhone case. `https://www.eff.org/deeplinks/2016/02/technical-perspective-apple-iphone-case`, February 2016.

[38] 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.

[39] 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.

[40] 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.

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

[42] Milan Broz and Others. dm-crypt. `http://code.google.com/p/cryptsetup/wiki/DMCrypt`.

[43] 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.

[44] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. RFC 1880: OpenPGP message format. Technical report, Internet Engineering Task Force, 2007.

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

[46] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano. The design and implementation of a transparent cryptographic file system for UNIX. In USENIX Annual Technical Conference, pages 199–212, 2001.

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

[48] Codenomicon. The Heartbleed bug. http://heartbleed.com.

[49] James B. Comey. Statement before the House Judiciary Committee. https://www.fbi.gov/news/testimony/encryption-tightrope-balancing-americans-security-and-privacy, March 2016.

[50] Commonwealth of Massachusetts Supreme Judicial Court. Commonwealth vs. Leon I. Gelfgatt. https://www.documentcloud.org/documents/1209519-commonwealth-vs-gelfgatt.html, June 2014.

[51] Tim Cook and Apple, Inc. A message to our customers. https://www.apple.com/customer-letter/, February 2016.

[52] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280: Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Technical report, Internet Engineering Task Force.

[53] Sophia Cope. Senate Judiciary Committee finally focuses on ECPA reform. https://www.eff.org/deeplinks/2015/09/senate-judiciary-committee-finally-focuses-ecpa-reform, September 2015.

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

[55] Douglas Crockford. Introducing JSON. http://www.json.org.

[56] Cryptography Contributors. Cryptography: A Python crypto library. https://cryptography.io/.

[57] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. Technical report, 1999.

[58] Xavier de Carné de Carnavalet and Mohammad Mannan. Challenges and implications of verifiable builds for security-critical open-source software. In Proceedings of the 30th Annual Computer Security Applications Conference, pages 16–25, New York, New York, USA, 2014. ACM Press.

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

[60] Anastasia Dergacheva and Ksenia Andreeva. Russia: The impact of new Internet regulations on international companies. Inside Counsel, September 2015.

[61] T. Dierks and E. Rescorla. RFC 5246: The transport layer security (TLS) protocol - version 1.2. Technical report, Internet Engineering Task Force, 2008.

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

[63] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In Proceedings of the 13th USENIX Security Symposium, 2004.

[64] 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.

[65] Dropbox, Inc. Dropbox. http://www.dropbox.com.

[66] Dropbox, Inc. Dropbox security. http://www.dropbox.com/security.

[67] Dropbox Inc. Dropbox transparency report. http://www.dropbox.com/transparency.

[68] Dropbox, Inc. Dropbox wasn't hacked. http://blog.dropbox.com/2014/10/dropbox-wasnt-hacked/.

[69] Dropbox, Inc. Yesterday's authentication bug. http://blog.dropbox.com/2011/06/yesterdays-authentication-bug/.

[70] Paul Ducklin. Bad news! LastPass breached. good news! you should be OK... Naked Security, June 2015.

[71] Morris Dworkin. Recommendation for block cipher modes of operation: The CMAC mode of authentication. Technical Report 800-38B, National Institute of Standards & Technology, 2005.

[72] P.J. Eby. PEP 3333 – Python web server gateway interface v1.0.1. Technical report, Python Software Foundation, 2010.

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

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

[75] Enigmail Project. Enigmail. http://www.enigmail.net.

[76] Facebook, Inc. Facebook. http://www.facebook.com.

[77] Facebook, Inc. United States law enforcement requests for data. https://govtrequests.facebook.com/country/United%20States/2015-H1/.

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

[79] Cyrus Farivar. Feds break through seized iPhone, stnad down in legal battle with Apple. Ars Technica, March 2016.

[80] Federal Bureau of Investigation. FOIA: National security letters. http://vault.fbi.gov/National%20Security%20Letters%20(NSL), 2007.

[81] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. Cryptography Engineering: Design Principles and Practical Applications. Wiley, Indianapolis, IN, 2010.

[82] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California Irvine, 2000.

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

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

[85] 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.

[86] 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.

[87] Ben Foster. How many users on Facebook. `www.benphoster.com/facebook-user-growth-chart-2004-2010/`, 2015.

[88] Free Software Foundation. GNU Affero General Public License. `https://www.gnu.org/licenses/agpl-3.0.en.html`. Version 3.0.

[89] Free Software Foundation. What is free software? `https://www.gnu.org/philosophy/free-sw.en.html`. Version 1.141.

[90] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jorg Schwenk, and Thorsten Holz. How secure is TextSecure? Cryptology ePrint Archive, 2014/904(February):17, 2014.

[91] Clemens Fruhwirth. LUKS. `https://code.google.com/p/cryptsetup`.

[92] Kevin E. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, 1998.

[93] Steven. 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.

[94] Sean Gallagher. EPIC fail: How OPM hackers tapped the mother lode of espionage data. Ars Technica, July 2015.

[95] Alberto Garcia and Martin Vigo. Even the LastPass will be stolen: Deal with it! Blackhat Europe, 2015.

[96] Gazzang. zTrustee. `http://www.gazzang.com/products/ztrustee`.

[97] 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.

[98] 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.

[99] Barton Gellman and Ashkan Soltani. NSA infiltrates links to Yahoo, Google data centers worldwide, Snowden documents say. The Washington Post, October 2013.

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

[101] GitHub, Inc. GitHub: How people build software. `https://github.com/`.

[102] Vindu Goel. Facebook tinkers with users' emotions in news feed experiment, stirring outcry. `http://www.nytimes.com/2014/06/30/technology/facebook-tinkers-with-users-emotions-in-news-feed-experiment-stirring-outcry.html`, June 2014. New York Times.

[103] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing remote untrusted storage. NDSS, (0121481), 2003.

[104] Y. Goland, W. Whitehead, A. Faizi, S. Carter, and D. Jensen. RFC 1528: HTTP extensions for distributed authoring – WEBDAV. Technical report, Internet Engineering Task Force, February 1999.

[105] 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.

[106] R.P. Goldberg. A survey of virtual machine research. IEEE Computer, 7(4):34–45, 1974.

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

[108] Dan Goodin. FBI: Silk Road mastermind couldnt even keep himself anonymous online. Ars Technica, 2013.

[109] Dan Goodin. How the Bible and YouTube are fueling the next frontier of password cracking. Ars Technica, 2013.

[110] Google, Inc. App engine. `http://cloud.google.com/appengine/docs`.

[111] Google, Inc. Compute engine. `http://cloud.google.com/compute/`.

[112] Google, Inc. Data center efficiency. `http://www.google.com/about/datacenters/efficiency/internal/`.

[113] Google, Inc. Docs. `http://docs.google.com`.

[114] Google, Inc. Drive. `http://drive.google.com`.

[115] Google, Inc. End-to-end. `http://github.com/google/end-to-end`.

[116] Google, Inc. Gmail. `https://mail.google.com`.

[117] Google, Inc. Google play music. `http://music.google.com`.

[118] Google Inc. Google transparency report. https://www.google.com/transparencyreport/userdatarequests/.

[119] Google, Inc. Hangouts. https://hangouts.google.com.

[120] Google, Inc. Plus. https://plus.google.com/.

[121] 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.

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

[123] Matthew Green. What's the matter with PGP? A Few Thoughts on Cryptographic Engineering, 2014.

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

[125] Glenn Greenwald and Ewen MacAskill. NSA Prism program taps in to user data of Apple, Google, and others. The Guardian, June 2013.

[126] Michael Austin Halcrow. eCryptfs. http://ecryptfs.org.

[127] Michael Austin Halcrow. eCryptfs : An enterprise-class cryptographic filesystem for Linux. In Ottawa Linux Symposium, pages 201–218, 2005.

[128] Kirk Hall. Certificate authority audits and browser root program requirements. CA Security Council, October 2013.

[129] HashiCorp. Vault: A tool for managing secrets. https://www.vaultproject.io/.

[130] Heroku. Heroku platform. http://www.heroku.com.

[131] Kashmir Hill. How Target figured out a teen girl was pregnant before her father did. Forbes, February 2012.

[132] 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.

[133] 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.

[134] 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.

[135] Inktank Storage. Ceph: The future of storage. http://ceph.com.

[136] 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.

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

[138] Michael K. Johnson. A tour of the Linux VFS. `http://www.tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html`, 1996.

[139] M. Jones, J. Bradles, and N. Sakimura. RFC 7515: JSON web signature (JWS). Technical report, Internet Engineering Task Force, May 2015.

[140] M. Jones, J. Bradles, and N. Sakimura. RFC 7519: JSON web token (JWT). Technical report, Internet Engineering Task Force, May 2015.

[141] JumpCloud. JumpCloud: Identity management reimagined. `http://jumpcloud.com`.

[142] 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.

[143] Seny Kamara, C Papamanthou, and Tom Roeder. CS2: A searchable cryptographic cloud storage system. Technical report, Microsoft Research, 2011.

[144] Danielle Kehl, Andi Wilson, and Kevin Bankston. Doomed to repeat history?: Lessons from the crypto wars of the 1990s. Technical Report June, New America, Open Technology Institute, 2015.

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

[146] Sanja Kelly, Madeline Earp, Laura Reed, Adrian Shahbaz, and Mai Truong. Freedom on the net 2015: Privatizing censorship, eroding privacy. Technical report, Freedom House, 2015.

[147] Gail Kent. The mutual legal assistance problem explained. Stanford Law School Center for Internet and Society Blog, February 2015.

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

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

[150] 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.

[151] Neal Koblitz. Elliptic curve cryptosystems. Mathematics of Computation, 48(177):203–209, 1987.

[152] Kagan Kocaoglu (Cahn Kojaolu). A comparative bibliography: Regulatory competition on corporate law. SSRN Electronic Journal, (March), 2008.

[153] Werner Koch. GnuPG. `http://www.gnupg.org`.

[154] Werner Koch and Marcus Brinkmann. STEED - usable end-to-end encryption. Technical report, 2011.

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

[156] David Kravets. FBI paid "grey hats" for zero-day exploit that unlocked seized iPhone. Ars Technica, April 2016.

[157] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104: HMAC - keyed-hashing for message authentication. Technical report, Internet Engineering Task Force, February 1997.

[158] 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.

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

[160] Brian Krebs. Data breach at health insurer Anthem could impact millions. Krebs on Security, February 2015.

[161] Brian Krebs. Premera Blue Cross breach exposes financial, medical records. Krebs on Security, Mar 2015.

[162] 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.

[163] LaCie AG. Wuala: Secure cloud storage. `http://www.wuala.com`.

[164] LastPass. LastPass password manager. `https://lastpass.com`.

[165] B. Laurie, A. Langley, and E. Kasper. RFC 6962: Certificate transparency. Technical report, Internet Engineering Task Force, 2013.

[166] P. Leach, M. Mealling, and R. Salz. RFC 4122: A universally unique identifier (UUID) URN namespace. Technical report, Internet Engineering Task Force, 2005.

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

[168] Least Authority. Simple secure storage service (S4). `http://leastauthority.com`.

[169] Micah Lee. Chatting in secret while we're all being watched. The Intercept, July 2015.

[170] Ladar Levison. Lavabit. `http://lavabit.com`.

[171] Ladar Levison. Secrets, lies and Snowden's email: Why I was forced to shut down Lavabit. The Guardian, May 2014.

[172] 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.

[173] Nicolas Lidzborski. Staying at the forefront of email security and reliability: HTTPS-only and 99.978% availability. Official Gmail Blog, Mar 2014.

[174] Linux-PAM Team. Linux PAM. http://www.linux-pam.org.

[175] LogRhythm, Inc. LogRhythm: SIEM 2.0, log and event management, log analysis. http://www.logrhythm.com.

[176] Natasha Lomas. Facebook data privacy class action joined by 11,000 and counting. TechCrunch, August 2014.

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

[178] Peter Loscocco and Stephen D Smalley. Meeting critical security objectives with security-enhanced Linux. In Ottawa Linux Symposium, 2001.

[179] Lyft. Confidant: Your secret keeper. https://lyft.github.io/confidant/.

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

[181] 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.

[182] Mailvelope GmbH. Mailvelope: OpenPGP encryption for webmail. https://www.mailvelope.com/.

[183] Moxie Marlinspike. The Cryptographic Doom Principle. Thought Crime, December 2011.

[184] Moxie Marlinspike. Advanced cryptographic ratcheting. Technical report, November 2013.

[185] 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.

[186] 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.

[187] David McGrew. Efficient authentication of large , dynamic data sets using Galois/counter mode (GCM). In Security in Storage Workshop. IEEE, 2005.

[188] Microsoft. Office Online. http://office.live.com.

[189] Microsoft. OneDrive. http://onedrive.live.com.

[190] Microsoft. Server message block (SMB) protocol versions 2 and 3. Technical Report MS-SMB2-46.0, 2014.

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

[192] Matthew Miller. Catch me if you can. Forbes, March 2006.

[193] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, Advances in Cryptology – CRYPTO '85 Proceedings, pages 417–426, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[194] 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.

[195] MIT Media Lab. OpenPDS software. Technical report, Human Dynamics, MIT Media Lab, 2012.

[196] Matt Monaco. Tutamen ask-password source code. https://git.monaco.cx/matt/tutamen-ask-password.

[197] Matt Monaco. Tutamen golang library source code. https://git.monaco.cx/matt/go-tutamen.

[198] Mozilla. Mozilla CA certificate policy: Version 2.2. https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/policy/.

[199] Randall Munroe. XKCD #504: Legal hacks. https://xkcd.com/504/, November 2008.

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

[201] National Institute of Standards & Technology. Data encryption standard. Technical Report 46, U.S. Dept. of Commerce, 1977. Federal Information Processing Standards Publication.

[202] National Institute of Standards & Technology. Announcing the advanced encryption standard (AES). Technical Report 197, U.S. Dept. of Commerce, 2001. Federal Information Processing Standards Publication.

[203] National Institute of Standards & Technology. Security requirements for cryptographic modules. Technical Report 140-2, U.S. Dept. of Commerce, 2001. Federal Information Processing Standards Publication.

[204] Ted Nelson. Computer Lib / Dream Machines. Self published, 1974.

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

[206] Neurio. Neurio home intelligence. http://www.neur.io.

[207] NightLabs Consulting GmbH. Cumulus4j. http://www.cumulus4j.org.

[208] nobody@jpunix.com. Thank you Bob Anderson: RC4 source code. http://cypherpunks.venona.com/archive/1994/09/msg00304.html.

[209] Greg Nojeim. MLAT reform: A straw man proposal. Center for Democracy and Technology Blog, September 2015.

[210] Donald A Norman. The design of everyday things. Basic Books, 2002.

[211] OASIS. Security assertion markup language (SAML). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security.

[212] OAuth Team. OAuth. http://oauth.net.

[213] Council of Insurance Agents and Brokers. Cyber Market Watch Survey. Technical report, April 2015.

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

[215] Open Whisper Systems. Open Whisper: Privacy that fits in your pocket. https://whispersystems.org.

[216] OpenPGP Alliance. OpenPGP. http://www.openpgp.org.

[217] OpenSSL. OpenSSL. http://www.openssl.org.

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

[219] OpenStack Project Team. Openstack. https://www.openstack.org.

[220] OpenVPN Technologies, Inc. OpenVPN: Your private path to access network resources and services securely. https://openvpn.net/.

[221] Kurt Opsahl. Warrant canary frequently asked questions. Technical report, Electronic Frontier Foundation, April 2014.

[222] Opscode. Chef. https://www.opscode.com/chef.

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

[224] OwnCloud Team. Owncloud server. http://owncloud.org.

[225] Jos Padilla and Jeff Lindsay. pyJWT: A Python implementation of RFC 7519. https://github.com/jpadilla/pyjwt.

[226] Jose Pagliery. Uber removes racy blog posts on prostitution, one-night stands. CNN Money, November 2014.

[227] 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.

[228] PCI Security Standard Council. PCI SSC data security standards overview. http://www.pcisecuritystandards.org/security_standards/index.php.

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

[230] James Pearson. Discovering secrets. changedmy.name, October 2013.

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

[232] 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.

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

[234] President's Review Group on Intelligence and Communications Technologies. Liberty and security in a changing world. December 2013.

[235] Puppet Labs. Puppet. http://puppetlabs.com.

[236] Rackspace. Cloud Keep. https://github.com/cloudkeep.

[237] John Rae-Grant. Making email safer for you. Official Gmail Blog, February 2016.

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

[239] Chet Ramey. GNU Bash. http://www.gnu.org/software/bash/.

[240] B. Ramsdell and S. Turner. RFC 5751: Secure/multipurpose internet mail extensions (S/MIME) version 3.2 message specification. Technical report, Internet Engineering Task Force, 2010.

[241] Redis Team. Redis: Remote dictionary server. http://redis.io/.

[242] Reporters Without Borders. The Internet in China. http://surveillance.rsf.org/en/china/, 2012.

[243] 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.

[244] 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.

[245] Armin Ronacher. Flask: Web development one drop at a time. http://flask.pocoo.org/.

[246] Alon Rosen. Analysis of the Porticor homomorphic key management protocol. Technical report, Porticor, 2012.

[247] Rebecca J. Rosen. Google Death: A tool to take care of your Gmail when you're gone. The Atlantic, April 2013.

[248] 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.

[249] RSA Laboratories. PKCS #11: Cryptographic token interface standard. http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm.

[250] Saltstack. Salt. http://www.saltstack.com.

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

[252] Jerome H. Saltzer and M.D. Schroeder. The protection of information in computer systems. Proceedings of the IEEE, 63(9):1278–1308, 1975.

[253] 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.

[254] 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.

[255] 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.

[256] 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.

[257] Andy Sayler. Custos server source code. https://github.com/asayler/custos-server.

[258] Andy Sayler. EncFS source code. https://github.com/asayler/custos-client-encfs.

[259] Andy Sayler. libcustos source code. https://github.com/asayler/custos-client-libcustos.

[260] Andy Sayler. pytutamen client source code. https://github.com/asayler/tutamen-pytutamen.

[261] Andy Sayler. pytutamen server source code. https://github.com/asayler/tutamen-pytutamen_server.

[262] Andy Sayler. Tutamen access control API source code. https://github.com/asayler/tutamen-api_accesscontrol.

[263] Andy Sayler. Tutamen command line interface. https://github.com/asayler/tutamen-tutamencli.

[264] Andy Sayler. Tutamen storage API source code. https://github.com/asayler/tutamen-api_storage.

[265] Andy Sayler. Custos: A flexibly secure key-value storage platform. Master's thesis, University of Colorado Boulder, December 2013.

[266] 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.

[267] Andy Sayler and Others. Tutamen-aware QEMU port. `https://github.com/asayler/qemu`.

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

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

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

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

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

[273] Bruce Schneier, Kathleen Seidel, and Saranya Vijayakumar. A worldwide survey of encryption products. SSRN Electronic Journal, 2:1–23, 2016.

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

[275] Claude Shannon. A mathematical theory of cryptography. Technical Report MM 45-11002, Bell Labs, 1945.

[276] Shibboleth Consortium. Shibboleth homepage. `http://shibboleth.net`.

[277] 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.

[278] Joe Siegrist. LastPass security notice. LastPass Blog, June 2015.

[279] Peter Sims. Can we trust Uber? `http://medium.com/@petersimsie/can-we-trust-uber-c0e793deda36`, September 2014. Medium.

[280] Abe Singer, Warren Anderson, and Rik Farrow. Rethinking password policies (uncut). ;login, 38(4), 2013.

[281] Simon Singh. The Code Book. Doubleday, 1999.

[282] Gregory C. Sisk. Official wrongdoing and the civil liability of the federal government and officers. University of St. Thomas Law Journal, 8(3), 2011.

[283] Craig Smith. How many people use 800+ of the top social networks, apps, and digital services. `http://expandedramblings.com/index.php/resource-how-many-people-use-the-top-social-media`, March 2005.

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

[285] Dag-Erling Smorgrav. OpenPAM. http://www.openpam.org.

[286] Daniel Solove. Going bankrupt with your personal data. Teach Privacy: Privacy and Security Blog, July 2015.

[287] J.H. Song, R. Poovendran, J. Lee, and T. Iwata. RFC 4493: The AES-CMAC algorithm. Technical report, Internet Engineering Task Force, June 2006.

[288] SpiderOak. SpiderOak: Store, sync, share, privately. http://spideroak.com.

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

[290] Square. Keywhiz: A system for managing and distributing secrets. https://square.github.io/keywhiz/.

[291] Tim Starks. Cyber insurance gets Hill attention. Politico, March 2016.

[292] Dan E Stigall. Ungoverned spaces, transnational crime, and the prohibition on extraterritorial enforcement jurisdiction in international law. SSRN, 2013.

[293] 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.

[294] Supreme Court of the United States. Near v. Minnesota. https://www.law.cornell.edu/supremecourt/text/283/697, 1931.

[295] Supreme Court of the United States. United States v. Miller. https://www.law.cornell.edu/supremecourt/text/307/174, 1939.

[296] Supreme Court of the United States. Katz v. United States. https://www.law.cornell.edu/supremecourt/text/389/347, 1967.

[297] Supreme Court of the United States. New York Times v. United States. https://www.law.cornell.edu/supremecourt/text/403/713, 1971.

[298] Supreme Court of the United States. United States v. Mitchell Miller. https://www.law.cornell.edu/supremecourt/text/425/435/, 1976.

[299] Supreme Court of the United States. Wooley v. Maynard. https://www.law.cornell.edu/supremecourt/text/430/705, 1977.

[300] Supreme Court of the United States. Smith v. Maryland. https://www.law.cornell.edu/supremecourt/text/442/735, June 1979.

[301] Supreme Court of the United States. United States v. Jones. https://www.law.cornell.edu/supremecourt/text/10-1259, 2012.

[302] 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.

[303] Peter Swire and Kenesa Ahmad. Goring dark versus a golden age for surveillance. Center for Democracy and Technology Blog, November 2011.

[304] Symantec. PGP. http://www.symantec.com/encryption.

[305] Symantec Corporation. 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.

[306] Symantec Corporation. Internet Security Threat Report 20. 20(April):119, 2015.

[307] systemd Contributors. systemd password agents. https://wiki.freedesktop.org/www/Software/systemd/PasswordAgents/.

[308] Miklos Szeredi and Others. FUSE: Filesystems in userspace. http://fuse.sourceforge.net.

[309] 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.

[310] Ken Thompson. Reflections on trusting trust. Communications of the ACM, 27(8):761–763, June 1984.

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

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

[313] 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.

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

[315] Twilio. Twilio: A messaging, voice, video and authentication API for every application. https://www.twilio.com/.

[316] Uber. Uber: Your ride, on demand. http://www.uber.com.

[317] United States. 28 U.S. Code, Title 28, Part V, Chapter 111 (Section 1651) - Writs. https://www.law.cornell.edu/uscode/text/28/1651.

[318] United States. 47 U.S. Code Subchapter I - interception of digital and other communications. https://www.law.cornell.edu/uscode/text/47/chapter-9/subchapter-I.

[319] United States. Federal deposit insurance corporation. https://www.fdic.gov.

[320] United States. Foreign intelligence surveillance court. www.fisc.uscourts.gov/.

[321] United States. Fifth Amendment to the U.S. Constitution. http://www.gpo.gov/fdsys/pkg/CDOC-110hdoc50/pdf/CDOC-110hdoc50.pdf, September 1789.

[322] United States. First Amendment to the U.S. Constitution. http://www.gpo.gov/fdsys/pkg/CDOC-110hdoc50/pdf/CDOC-110hdoc50.pdf, September 1789.

[323] United States. Fourth Amendment to the U.S. Constitution. http://www.gpo.gov/fdsys/pkg/CDOC-110hdoc50/pdf/CDOC-110hdoc50.pdf, September 1789.

[324] United States. Second Amendment to the U.S. Constitution. http://www.gpo.gov/fdsys/pkg/CDOC-110hdoc50/pdf/CDOC-110hdoc50.pdf, September 1789.

[325] Unknown. Popcorn Time: Watch movies and TV shows instantly. http://popcorntime.io.

[326] U.S. Court of Appeals for the Eleventh Circuit. United States v. John Doe. https://www.eff.org/files/filenode/opiniondoe22312.pdf, 2012.

[327] U.S. Court of Appeals for the Ninth Circuit. Daniel J. Bernstein et al., v. United States Department of State et al. https://www.epic.org/crypto/export_controls/bernstein_decision_9_cir.html, May 1999. 97-16686.

[328] U.S. Court of Appeals for the Sixth Circuit. Peter D. Junger v. William Daley, U.S. Secretary of Commerce, et al. http://caselaw.findlaw.com/us-6th-circuit/1074126.html, April 2000. 98-4045.

[329] U.S. Department of Commerce: Bureau of Industry and Security. Identifying encryption items. https://www.bis.doc.gov/index.php/policy-guidance/encryption/identifying-encryption-items.

[330] U.S. Department of Education. Family Educational Rights and Privacy Act. https://www2.ed.gov/policy/gen/guid/fpco/ferpa/index.html.

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

[332] U.S. Department of Justice, Office of Justice Programs, Bureau of Justice Assistance. Electronic Communications Privacy Act of 1986 (ECPA), 18 U.S.C. Section 2510-22. https://it.ojp.gov/privacyliberty/authorities/statutes/1285.

[333] U.S. District Court for the District of Vermont. In re Boucher. http://www.volokh.com/files/Boucher.pdf, 2007.

[334] U.S. Federal Communication Commission. Communications assistance for law enforcement act. https://www.fcc.gov/public-safety-and-homeland-security/policy-and-licensing-division/general/communications-assistance.

[335] U.S. Federal Trade Commission. ASUS settles FTC charges that insecure home routers and cloud services put consumers' privacy at risk. https://www.ftc.gov/news-events/press-releases/2016/02/asus-settles-ftc-charges-insecure-home-routers-cloud-services-put, February 2016.

[336] U.S. House of Representatives: 114th Congress. H.R. 4528: ENCRYPT Act of 2016. https://homeland.house.gov/wp-content/uploads/2016/03/2016.03.03_HR-4651-Commission.pdf.

[337] U.S. House of Representatives: 114th Congress. H.R. 4651: Digital Security Commission Act of 2016 and Technology Challenges. https://homeland.house.gov/wp-content/uploads/2016/03/2016.03.03_HR-4651-Commission.pdf.

[338] U.S. Office of Personnel Management. Cybersecurity resource center: Cybersecurity incidents. https://www.opm.gov/cybersecurity/cybersecurity-incidents/.

[339] U.S. Senate: 114th Congress. Compliance with Court Orders Act of 2016. http://www.feinstein.senate.gov/public/index.cfm/press-releases?ID=EA927EA1-E098-4E62-8E61-DF55CBAC1649, April 2016.

[340] Mohammad Hadi Valipour, Bavar Amirzafari, Khashayar Niki Maleki, and Negin Daneshpour. A brief survey of software architecture concepts and service oriented architecture. In 2009 2nd IEEE International Conference on Computer Science and Information Technology, pages 34–38. IEEE, 2009.

[341] Kurt Wagner. How Facebook is using your photos in ads. Mashable, September 2013.

[342] Ian Walden. Accessing data in the cloud: The long arm of the law enforcement agent. SSRN Electronic Journal, 2004(185):1–24, 2011.

[343] WhatsApp, Inc. WhatsApp: Simple, personal, real time messaging. https://www.whatsapp.com/.

[344] White House Office of the Press Secretary. Statement of the press secretary. https://www.epic.org/crypto/clipper/white_house_statement_2_94.html, February 1994.

[345] 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.

[346] 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.

[347] Duane Wilson and Giuseppe Ateniese. To share or not to share in client-side encrypted clouds. arXiv:1404.2697, November 2014.

[348] World Wide Web Consortium (W3C). W3C homepage. https://www.w3.org/.

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

[350] Tatu Ylonen. SSH-secure login connections over the Internet. Proceedings of the 6th USENIX Security Symposium, 1996.

[351] Philip Zimmermann. PGP Source Code and Internals. MIT Press, 1995.

[352] Philip Zimmermann. PGP marks 10th anniversary. https://www.philzimmermann.com/text/PGP_10thAnniversary.txt, June 2001.