

Protecting User Data in Large-Scale Web Services

by

Frank Yi-Fei Wang

B.S., Stanford University (2012)

S.M., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

Author.....

Department of Electrical Engineering and Computer Science

June 25, 2018

Certified by

Nickolai Zeldovich

Professor

Thesis Supervisor

Certified by

James Mickens

Associate Professor

Thesis Supervisor

Accepted by

Professor Leslie A. Kolodziejski

Professor of Electrical Engineering and Computer Science

Chair, Department Committee on Graduate Students

Protecting User Data in Large-Scale Web Services

by

Frank Yi-Fei Wang

Submitted to the Department of Electrical Engineering and Computer Science
on June 25, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Thesis Supervisor: Nickolai Zeldovich
Title: Professor

Thesis Supervisor: James Mickens
Title: Associate Professor

Acknowledgments

Contents

1	Introduction	11
1.1	The Challenges of Centralized Data	12
1.2	Our Solution	13
2	Threat Model and Security Goals	17
2.1	Threat Model	17
2.2	Security Goals	18
3	System Design	21
3.1	System Components	21
3.2	Usage Model	22
3.3	Overview of ABE	23
3.4	Assigning Attributes to User Data	24
3.5	Uploading Data to the Storage Provider	25
3.6	Defining and Enforcing Access Policies	26
3.7	Key Revocation	27
3.7.1	Re-encrypting data	28
3.7.2	Re-encrypting metadata	30
3.7.3	Additional details	31
3.8	Protecting Against Device Loss	32
3.9	Minimizing ABE Overheads	33
3.10	Relabelling	35
3.11	Discussion	36

3.11.1	Alternative policy languages	36
3.11.2	Paying for storage	37
3.11.3	Efficient data importing	37
3.11.4	Anonymity across services	38
4	Implementation	39
5	Evaluation	41
5.1	Case Studies	43
5.1.1	Open mHealth	43
5.1.2	Piwigo	44
5.1.3	Server-side per-core throughput	44
5.2	Microbenchmarks	45
5.2.1	Encryption speed	45
5.2.2	Key generation and revocation	46
5.2.3	Attribute matching	46
5.2.4	Secret-sharing	47
6	Related Work	49
6.1	Untrusted servers	49
6.2	ABE-protected storage	50
6.3	Predicate encrypted storage	51
6.4	Access delegation schemes	51
7	Conclusion and Future Work	53
7.1	Conclusion	53
7.2	Future Work	53

List of Figures

1-1	Sieve’s high-level architecture	13
1-2	Example Sieve policy	14
3-1	ABE Example	23
3-2	Example of Sieve storage-based data structure	34
5-1	ABE throughputs for Sieve	42
5-2	Computational overheads for key generation and revocation.	46

Chapter 1

Introduction

In today's web, a single person often uses multiple web services. Conceptually, the user has a single logical data set, and she selectively exposes a portion of that data to each web service. In practice, the services control her data: each service keeps its portion of the user's objects in a walled garden which neither the user nor external services can directly access.

Although currently, the user can use access control lists or adjust her privacy settings to restrict data sharing for different web services, there are several problems. First, the user has to trust the cloud provider will honor these settings. Moreover, different providers might have different ways to set privacy policies, and these policies might not be sufficiently expressive. With these service-controlled data silos, users cede the ultimate power to set access controls and limit how their data is shared. Furthermore, simple operations like enumerating all of a user's cloud data become difficult, since a user's state is scattered across a variety of services which hide raw storage via high-level, curated APIs.

Data silos are not only problematic for users but also problematic for *applications* whose value often scales with the amount of user data that can be accessed by the application. For example, quantified self applications [?], which track a user's health and personal productivity, work best when given data from a variety of sensors and environmental locations. Similarly, applications which analyze a user's medical records [?] or financial transactions [?] produce the best results when they have access to all of the user's medical or financial data. Unfortunately, modern web services use a siloed design that limits a user's ability to share data outside of the service-specific silo. For example, wearable

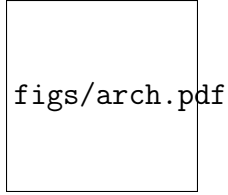
fitness-tracking sensors typically upload data to vendor-specific cloud storage, and medical records are often bound to storage that belongs to the medical specialist who measured the data. Thus, having a user-centric storage model is beneficial for *both* the user and the application. However, using such a centralized model poses new challenges.

1.1 The Challenges of Centralized Data

In a user-centric storage model, a user's entire data set would reside in a single, logically centralized cloud store; the user would selectively disclose portions of that data to individual third party applications. Systems like Amber [?] and BStore [?] explored the potential benefits of decoupling applications and user data, but a centralized data storage increases the damage that results from a subverted or curious storage provider because *all* of a user's cleartext data is at risk, instead of a service-specific subset.

To protect against untrusted storage providers, users can encrypt data before uploading it. However, the ultimate purpose of uploading data is to share it with third party services. Thus, users need a way to selectively expose pointers to encrypted objects (and the associated decryption keys). Protocols exist for sharing cloud data across multiple services, but these protocols have major usability and security problems. For example, the popular OAuth protocol [?] enables cross-site data sharing via delegated API calls (i.e., API calls that act with the authority of a user). However, OAuth policies are invariably defined by web services, not by users. Furthermore, OAuth does not enforce cryptographically strong constraints on the data that the delegated APIs can access. Thus, even if a user could generate her own OAuth policies, she would lack strong assurances about what those policies mean, and how they are enforced.

Given the discussion above, logically centralized storage seems good for users in theory, but difficult to implement in practice. In this thesis, we address three specific challenges that emerge from a logically centralized storage architecture. The first is *security*: how do we make access controls cryptographically strong, and protect user data against the compromise of storage providers and user devices? The second challenge is *usability*: how can we express access policies in a way that lay-person users will understand, but is translatable



figs/arch.pdf

Figure 1-1: Sieve’s high-level architecture. 1) The user uploads ABE-encrypted data to a storage provider. 2) The user generates a data policy for a third party web service. Sieve translates the policy into an ABE decryption key, and sends the key to the web service. 3) The web service pulls encrypted data from the storage provider, decrypts it locally, and injects the data into the unmodified application pipeline.

to cryptographically enforceable mechanisms? The final challenge is *application richness*: after we have moved user data out of per-application silos and into user-controlled storage, how can we support the complex applications that users currently enjoy?

It is important acknowledge that in this system, we focus primarily on solving the security challenges with centralized data. Performance challenges that result from using centralized data, such as the challenges of playing videos efficiently from remote storage, are out of scope of this thesis.

1.2 Our Solution

To address these challenges, we propose Sieve, a new system for delegating access to private cloud data. Figure 1-1 depicts Sieve’s high-level workflow. A user generates raw data on her computational devices, and uploads encrypted versions of it to a single cloud repository; the user manages and pays for the cloud, not the web services which desire access to the data. When a third party web service requests access, e.g., during the first time that a user visits a site, the user generates a high-level access policy (§3.6) for the service. Sieve splits the policy into two pieces: the storage provider learns which objects a third party can access (but not the cleartext versions of those objects), and the third party learns the objects that it can access, and the corresponding decryption keys, while learning nothing about the rest of the user’s data set. Once the third party has downloaded the necessary objects and decrypted them, it feeds the cleartext data to a legacy pipeline for processing user content (§3.4).

Sieve leverages three techniques to implement the workflow in Figure 1-1:

(type="Fitness" OR type="Medical") AND (date > 2012)
AND (source="FitBit")

Figure 1-2: Example Sieve policy for an exercise application.

- Sieve uses **attribute-based encryption** (ABE) [?] to implement cryptographically strong access policies. In ABE, encrypted data is associated with attributes, which are key-value pairs like “date=2012”, and decryption keys are associated with policies; a key can only decrypt whose attributes satisfy a key’s policy. The policies and attributes are both stored in plaintext. Before the user uploads objects to the storage provider, she (or her local device) tags the objects with attributes like the date, the user’s current location, or the object type (e.g., “medical”, “financial”, “fitness”). The uploading device transparently encrypts the objects with the relevant attributes before sending the objects to the storage provider. Later, when a third party web service requests access to the user’s data, the user creates a policy defined in terms of attributes and attribute operators like equality and less than. Figure 1-2 shows an example of such a policy, which a user might give an application that visualizes her exercise data. The user’s local device automatically translates the policy into an ABE decryption key, and sends the key to the web service. Afterwards, the web service can use the key to read the cleartext versions of the delegated objects from the storage provider and does not require any more communication with the user.
- To revoke a third party’s access to sensitive data, the user informs the storage provider that the third party should no longer be able to download encrypted user objects. However, the third party still possesses the associated decryption key, and can decrypt leaked ciphertext if the storage server is later compromised. To prevent this scenario, Sieve uses **key homomorphism** [?] to implement key revocation. Key homomorphism allows the storage provider to re-encrypt user data without learning the underlying cleartext—the storage provider merely reads the ciphertext and writes the output of a function that accepts the ciphertext and a user-specified rekeying token as input. The execution of this function on the storage provider also does not

reveal any information about the cleartext data. Using *in situ* re-encryption, users can avoid re-encrypting data on user devices and then re-uploading it. Additionally, if storage providers are honest at the time of key revocation, subsequent storage provider compromises will not reveal data that is encrypted with keys that are revoked (but possibly still in the wild). To our knowledge, Sieve is the first ABE-based system to support full re-keying of *both* the metadata and data.

- ABE uses a *master secret key* to generate decryption keys. The loss of this key results in the compromise of the entire cryptosystem. In standard ABE schemes, the master secret is kept by a single trusted authority. In the context of Sieve, this would mean keeping the master secret on a single user device. This is unattractive, since user devices are often lost or stolen. Thus, Sieve uses **secret-sharing** and **two-factor authentication** to partition the master secret across multiple devices, and prevent unauthorized devices from arbitrarily participating in Sieve’s cryptographic protocols.

Sieve represents a middle ground between today’s popular web services (which provide weak user control over the distribution and access of her data), and proposed systems from the research community which strengthen user control, but greatly restrict server-side computation [?] or eliminate it altogether [? ? ? ? ?]. Sieve explores a different point in the design space, one that provides cryptographically strong access controls to users while still permitting the rich server-side computation on user’s sensitive data that many popular web applications require to provide increased value for the user. More specifically, Sieve works best with applications where data streams are bound to the user and services that primarily read from data stream rather than write to them. Applications, like Reddit [?], that have data that has little standalone value but derives value primarily from being part of a larger context, like a discussion forum, are less applicable to Sieve.

To demonstrate its practicality, we integrated Sieve with two different types of applications, a health application, Open mHealth [?], and a photo application, Piwigo [?]. Open mHealth is an open-source application that helps users store and visualize their data. We obtained realistic user health data based on existing clinical standards from Open mHealth and used Sieve to manage the user data (§5.1). Piwigo is an open-source photo application that allows users to store and manipulate their photos similar to Flickr [?]. Like Open

mHealth, we used Sieve to manage the photos in Piwigo. Integrating Sieve into the Open mHealth and Piwigo was straightforward and required approximately 200 and 250 lines of code respectively. The experimental results show these integrated systems can handle realistic workloads.

Chapter 2

Threat Model and Security Goals

We focus on three kinds of principals. A **user** is someone who wants to store data online and selectively expose it to a **third party web service**. The user stores her (encrypted) data on a **cloud storage provider**. Each user has one storage provider, but potentially many third parties which need delegated access. We envision storage providers be operated by services such as Amazon S3 and Microsoft Azure. Potential third party web services are FitBit, Lark, Misfit, Mint, and any other application that generates new value from sensitive user data.

2.1 Threat Model

Sieve protects against a curious cloud provider or external attacker from gaining access to user data stored on the cloud storage provider. We assume the attacker is *passive*, meaning they can access the data but do not interfere in any Sieve protocols.

Sieve prevents the web service from trying to access more data than allowed by its access policies. Moreover, Sieve allows for full revocation of a web service's data access: after revocation, the web service cannot access any previously retrievable data even if the storage provider is compromised.

The user is trusted, but Sieve assumes that the user might lose devices. Sieve provides a way to recover from device losses, but Sieve does not protect against a subverted device that the user believes is functioning properly, i.e. a device installed with a rootkit.

Sieve does not protect against client-side attacks like social engineering [?] or cross-site

scripting [?].

2.2 Security Goals

The user has a financial agreement with the storage provider: the user pays for the provider to keep her objects and to participate in the Sieve protocol on her behalf. The user places encrypted data on the storage provider, but never reveals the decryption keys to the provider. This protects the confidentiality of user data if the storage service is malicious or compromised. Using signatures, Sieve protects the data's integrity. However, Sieve cannot guarantee the availability or freshness of the data that the storage provider delivers to a web service. If desired, Sieve can be layered atop storage systems like CloudProof [?] which do provide those properties.

Sieve does not hide access patterns or object metadata (i.e., attributes) from the storage provider. Thus, a curious provider can learn which encrypted objects a third party has been authorized to read, as well as the attributes that are associated with those objects. If users are concerned about data leakage via access patterns, they could layer Sieve atop an ORAM protocol [?]. To hide attributes, Sieve could also use predicate encryption [? ?]. However, ORAM and predicate encryption incur heavy performance overheads (§6), so Sieve uses lighter weight mechanisms that reduce service latency at the cost of leaking more metadata. We believe that this trade-off is reasonable for many users and companies, given the importance of low latencies in modern web services [? ?].

With respect to third party web services, Sieve's goal is to reveal user data only as permitted by the user's disclosure policies. After Sieve transmits information to a third party web server, Sieve cannot restrict what the third party does with that data, since third parties may cache user data locally, even after the user has revoked access to the canonical versions that reside on her storage server. Third parties can also share decrypted user data with other principals via out-of-band, non-Sieve protocols like in the current web infrastructure. Solving these issues is beyond the scope of this paper. However, if a web service shares its user-issued ABE key, Sieve can revoke it, preventing those parties from downloading old or new user data from the storage provider (§3.7).

Sieve uses secret sharing to protect system-wide secrets like the ABE master key from the loss of a single device (§3.8).

Chapter 3

System Design

This chapter will discuss the design details of Sieve. First, we will give an overview of the major system components. Next, we will discuss the best usage models for Sieve. We will provide an overview of ABE and how to assign attributes to user data. Then, we will talk about how data is uploaded to the storage provider. We will discuss how to define access policies and issue keys to web services. We will provide a key revocation scheme and talk about protecting against device loss. We, then, present a technique to minimize ABE overheads. Finally, we show how to change attributes on data in the storage provider.

3.1 System Components

As shown and described in Chapter 1, Sieve consists of three components: a user client, a storage provider daemon, and an import daemon run by third party web services. The **Sieve client** runs on each user device. The client implements a Javascript library which allows users to define the access policies for each web service (§3.6). For each policy, the client generates the associated ABE decryption key, and sends it to the appropriate web service. Before the user uploads an object to the storage provider, the Sieve client transparently signs the object, and then encrypts the object and signature with the relevant ABE attributes (§3.4). To protect against lost or stolen devices, the RSA signing key and the ABE master secret are partitioned across user devices; before a client can sign or encrypt an object, the client must participate in a secret sharing protocol with two-factor authentication (§3.8).

The **storage provider daemon** accepts encrypted data from Sieve clients. Each encrypted object is tagged with cleartext attributes a_0, \dots, a_N , where a_i is an attribute name and value, such as “name=Joe”. The storage provider builds an index of these attributes so that, when a web service requests objects whose attributes satisfy a certain policy, i.e. the policy on their ABE decryption key, the storage provider can efficiently locate those objects (§3.4).

The **Sieve import daemon** is run by a third party web service. The import daemon accepts ABE keys and upload notifications from Sieve clients. After receiving an upload notification, the import daemon fetches encrypted objects from the storage provider. The import daemon then decrypts the objects using its ABE key, and feeds the cleartext data into the web service’s legacy pipeline for consuming user data (§3.4).

From the perspective of a third party, a user’s Sieve storage is read-only; only the user can write new objects and update old ones. Third parties use their own storage for data that is derived from a user’s Sieve objects.

In Sieve, there are five types of keys. There is the user **ABE master key**, which is used to derive **ABE decryption keys**. The **ABE public key** is used to encrypt the metadata block (§3.5) under user-provided attributes. The ABE decryption keys, which are associated with a policy and given to web services by the user, are used to decrypt metadata blocks. The actual data are signed with **per-user RSA keys** and encrypted with **symmetric keys**.

3.2 Usage Model

In theory, any web service that imports user data is compatible with Sieve. In practice, certain kinds of web services and user data are easier to integrate into the Sieve model. Sieve works best with

- data streams that are tightly bound to a particular user, and
- web services that can tolerate those data streams being read-only (and perhaps only partially disclosed).

Examples of Sieve-amenable data streams include demographic information like age and location; sensitive financial and medical records; sensor data from quantified self applications;

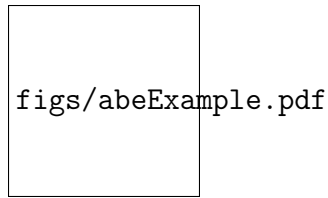


Figure 3-1: In this example, there are two ABE keys at the top, and three ABE-encrypted objects at the bottom. An arrow indicates that a key can decrypt a particular object.

longitudinal, cross-site histories of browsing behavior and e-commerce transactions; and multimedia data streams containing photos, videos, and audio. Examples of web services that consume such data streams are social media applications like Instagram [?], exercise trackers like Open mHealth [?], and financial analysis sites like Mint [?] that require access to a user's spending habits.

Applications like Reddit [?] and StackOverflow [?] are less appropriate for Sieve. In these applications, user data has less standalone value to the owner; instead, most of the value derives from embedding the data in a larger, service-specific context like a Reddit discussion. Web-based email is also an awkward fit for the Sieve model, since email services require mutable, per-user state like mailboxes, but Sieve exports read-only storage. An email service could pull read-only outgoing messages from Sieve storage, and implement the mutable mailbox state on the service's own machines. However, such an architecture would be awkward, since users would have no way to selectively expose *incoming* messages to the mail service.

3.3 Overview of ABE

Attribute-based encryption [?] is a public-key encryption scheme. Each user has a unique ABE master key, which is used to derive ABE decryption keys. A cleartext object is associated with attributes that govern how the object is encrypted. The ABE public key used to encrypt the object is derived directly from these attributes, which can be integers or strings. Since ABE is a public-key encryption scheme, anyone can encrypt objects. In Section 3.10, we discuss how to change these labels after encryption. Each decryption key has an access control structure (ACS) which enumerates one or more attributes. An ACS

can test an attribute for equality (e.g., `location='Paris'`) or comparative value (e.g., `age > 35`). An ACS can also chain those simple tests using ANDs, ORs, and NOTs. As shown in Figure 3-1, a key can only decrypt an object if the key's ACS matches the object's attribute values. Using this user-generated ACS, the user derives these ABE decryption keys from the ABE master key. Finally, since ABE is a public-key encryption scheme, it is slower than symmetric key cryptography.

We use the shorthand notation K_{a_0, \dots, a_N} to refer to an ABE key whose ACS contains the attribute tests a_0, \dots, a_N where a_i is an attribute name and value. All tests are implicitly joined via ANDs unless we explicitly note otherwise.

3.4 Assigning Attributes to User Data

Raw user data comes from a variety of sources. Some of it is directly generated by a user's devices; for example, a user might enter financial information directly into a spreadsheet. Data might also come from an external source, like an email attachment. Sieve associates each object, regardless of its provenance, with a set of attributes.

Some attributes can be automatically generated by hardware, like the GPS coordinates for a running route. Other attributes can be extracted by software, using application-specific transducers or semantic file systems [? ? ?]. Users can also manually tag objects. Sieve is agnostic to the manner in which attributes are created, although our implementation of the Sieve client provides a GUI which simplifies manual tag assignment. Users can also change the data's attributes after encryption (§3.10).

Users and web services must agree on data schemas, so that web services can meaningfully aggregate and process information from different users. In particular, web services need to know a standardized set of attributes which are defined by user data. To define this standardized set, Sieve uses FOAF [?] as the schema model for data about human users, and RDF [?] as the schema model for data about user objects.

Each Sieve user has a standardized FOAF record which stores basic scalar information like her name, location, and birthday. Sieve associates each entry in the FOAF record with an ABE attribute; for example, a user's name is associated with the *userName* attribute. Sieve

does not upload the actual FOAF record to the storage provider, since the only purpose of the FOAF record is to standardize the metadata that is associated with each user. Instead, Sieve uploads individual FOAF entries, encrypting each one with the associated ABE attribute (e.g., $\langle \text{alice} \rangle_{K_{\text{field=userName}}}$, where “alice” is the encrypted data and $K_{\text{field=userName}}$ means that the data can only be decrypted by web services whose ABE keys possess the access to the *userName* field attribute).

Sieve associates each data object with a type-specific RDF schema. For example, a W2 tax record has attributes for the user’s pre-tax income, her number of dependents, and so on. Similar to FOAF records, RDF records are only used to define a standard attribute set for each data type. Sieve uploads individual encrypted RDF entries to the storage provider.

Some data objects like images are not decomposable, i.e., each object is disclosed or not disclosed at the granularity of the entire object. For objects like this, Sieve uploads the entire object, encrypting it using the standardized RDF attributes and any manually added user tags. For example, a photo has standard attributes like height and width, and may also possess user-defined tags like “Vacation” or “Family.”

As applications evolve, RDF and FOAF schemas may change. Sieve is compatible with preexisting techniques for synchronizing schema changes across a distributed system [? ? ?].

3.5 Uploading Data to the Storage Provider

Suppose that the user wishes to upload a file F that has attributes a_0, \dots, a_N . Before uploading the file, the Sieve client must encrypt the file such that decryption is only possible with the ABE key whose ACS match the attributes a_0, \dots, a_N . The naïve approach is to directly encrypt F with K_{a_0, \dots, a_N} . Since ABE is significantly slower than symmetric key cryptography. Thus, Sieve uses a *hybrid encryption* scheme, encrypting the file data with a symmetric key, and encrypting the symmetric key with ABE.

The end-to-end upload protocol is the following. First, Sieve generates a symmetric key k , and uses that key to encrypt F . Sieve uploads the encrypted $\langle F \rangle_k$ to the storage provider. The storage provider responds with a GUID for the file. To reduce latency, a user

can also request and cache available GUIDs before the Sieve protocol starts to avoid having to wait for a server response for each file upload. Sieve then uploads a *metadata block* for F . The metadata block is an encrypted pointer containing $\langle GUID, k \rangle_{K_{a_0, \dots, a_N}}$. Only principals which possess keys with ACSes that match a_0, \dots, a_N can decrypt the pointer, fetch the object, and decrypt the object.

In the next section, we describe how web services acquire ABE keys. For now, we merely say that users do not share ABE keys with storage providers. Thus, a storage provider cannot inspect the data that it stores. The provider can try to tamper with the data or produce fake user data, but Sieve clients sign each object with a user-specific RSA key before encrypting the object and its signature, allowing web services to detect tampering.

From the perspective of a third party web service, Sieve storage is read-only. However, a user is free to create new objects, delete old ones, and update objects that reside at preexisting GUIDs. If a user's device has cached the GUID and the symmetric key for a particular object, the user can update that object directly, without having to fetch the associated metadata block and incur ABE overheads to decrypt it.

3.6 Defining and Enforcing Access Policies

In Sieve, all user data is private by default, since users must explicitly share ABE decryption keys that provide access to data. When a third party requests access permissions, e.g., upon the first time that a user visits a site, the user generates an access policy for the site. Policies are defined in terms of attributes, and the Sieve client provides a GUI which makes it easy for users to explore which attributes her data contains, and which objects would be exposed for a given policy. Policies are simple boolean expressions; for example, a web service used by a physician might receive the policy (fileType="medicalRecord" AND year>2010 AND doctor="John").

After the user has defined a policy, her Sieve client assembles the ABE master secret (§3.8) and generates an ABE key with the appropriate ACS. The Sieve client then sends the key and the name of the user's storage provider to the remote web service. The message is protected with SSL/TLS [?] to ensure confidentiality and integrity. Later, when the

web service desires to access user data, the service does not need to interact with the user. Instead, the service sends an access request directly to the user’s storage provider. The request contains a list of the attributes which belong to the data of interest. The storage provider returns the encrypted metadata blocks for the relevant objects. The web service decrypts the metadata, revealing the GUIDs for the requested objects as well as their symmetric encryption keys. The web service uses the GUIDs to fetch the encrypted objects. After decrypting the objects locally, the service feeds the cleartext data into an application-specific data pipeline.

Once that happens, Sieve is uninvolved in the application workflow. Thus, Sieve is compatible with the current web ecosystem which uses third party computation and storage to add value to user data. However, Sieve provides users with cryptographically strong control over the raw data that each service receives. Sieve’s access policies also have three attractive properties:

- The number of policies (and keys) scales with the number of web services that a user shares with, not the much larger number of objects that she owns.
- Policy generation is decoupled from object generation. At object creation time, users do not have to speculate a priori about whom a new object might be shared with.
- Policies safeguard objects using cryptography, but users are insulated from the details of key management.

Given all of this, we believe that Sieve strikes a good balance between security, usability, and backwards compatibility with current web services.

3.7 Key Revocation

In Sieve, an individual object is encrypted with a symmetric key k ; the object’s metadata block (which contains k and the object’s GUID) is encrypted with an ABE key K_{a_0, \dots, a_N} . A web service caches its ABE key, and it may also cache symmetric keys and GUIDs, to avoid repeated fetches and decryptions of metadata blocks. Caching makes revocation tricky, since a user that wants to revoke a service’s access rights cannot force the service to delete cached keys. An honest storage provider can refuse access requests from deprivileged third parties,

but if the storage provider is compromised, it can leak data that is encrypted with ostensibly revoked keys that are still in the wild.

To protect against storage server compromise, Sieve revokes keys by *re-encrypting* user data and metadata with new keys that are not shared with the newly deprivileged third party. If the storage provider is honest at the time of rekeying, then even if it is compromised later, it will never leak data that is encrypted with revoked keys.

Leveraging homomorphic encryption [?], the storage provider re-encrypts the data locally, using a rekeying token provided by the user. The storage provider learns nothing about the old encryption key, the new encryption key, or the underlying cleartext; the user avoids the need to download, re-encrypt, and re-upload data from her personal devices.

In the rest of this section, we first describe how data is re-encrypted, and then explain how the metadata is re-encrypted.

3.7.1 Re-encrypting data

To enable storage providers to re-encrypt data in situ, Sieve employs a key homomorphic pseudorandom function [? ?]. The intuition is that we use this pseudorandom function in place of the AES block cipher for counter mode encryption. With the properties of counter mode and a key homomorphic pseudorandom function, we can construct a key homomorphic encryption scheme that allows Sieve to re-key data without decrypting.

We define that function F as

$$F(k, x) = H(x)^k$$

where H is a hash function and k is the secret key associated with each object x is the input into the function. F is additively key homomorphic, which means that, for two keys k and k' , $F(k, x) \cdot F(k', x) = F(k + k', x)$. All operations described in this section are done modulo p where p is a large prime.

Using F , we define an encryption scheme whose security is similar to those of AES. We discuss the security of this scheme more formally below. Like AES-CTR, our new encryption scheme operates on blocks of data, and uses a random counter to convert a block

cipher into a stream cipher. In our new scheme, the j th ciphertext block c_j is equal to

$$c_j = m_j \cdot F(k, N + j)$$

where m_j is the j th cleartext block, and N is a public nonce that is equivalent to the initialization vector in AES-CTR. To decrypt, a third party extracts k from a metadata block and performs the following calculation:

$$m_j = c_j \cdot F(-k, N + j)$$

To revoke the ABE key K_{a_0, \dots, a_N} , a user's Sieve client generates a *rekeying token* for each object that is accessible via K_{a_0, \dots, a_N} . For an object encrypted by k , the rekeying token is $\delta = -k + k'$, where k' represents the new encryption key for the object. The client sends δ to the storage provider; this operation is safe because the provider cannot recover k or k' from δ . The storage provider uses δ to compute a new version of each ciphertext block c_j :

$$\begin{aligned} c_{j, \text{new}} &= c_j \cdot F(\delta, N + j) \\ &= m_j \cdot F(k, N + j) \cdot F(-k + k', N + j) \\ &= m_j \cdot F(k', N + j) \end{aligned}$$

In this manner, the storage provider re-encrypts objects without learning the encryption keys or the underlying cleartext.

Security for key homomorphic encryption scheme: We discuss some security goals and properties of the key homomorphic encryption scheme described above. We want to preserve the semantic security of AES-CTR, a common symmetric key encryption scheme. We will show our key homomorphic encryption scheme is also semantically secure.

First, Naor, Pinkas, and Reingold [?] showed that $F_{NPR}(k, x) = H(x)^k$ (described above) is a secure pseudorandom function (PRF) in the random oracle model [?] if the Decision Diffie-Hellman assumption holds in \mathbb{G} . An additional feature of this PRF is that it is key homomorphic, but that additional property does not affect the security of the PRF.

Now that we have a secure PRF. We define counter-mode encryption as $c_j = m_j \cdot F(k, N + j)$ where c_j is the j th block of the ciphertext, m_j is the j th block of the plaintext, k is the symmetric key, and N is a unique nonce. A survey by Rogaway [?] has shown that this encryption scheme is semantically secure as long as F is a secure PRF. Typically, the AES block cipher is used as F because AES is assumed to be a secure PRF. In our encryption scheme, we use F_{NPR} as our PRF, which is key homomorphic.

Therefore, our encryption scheme is semantically secure in the random oracle model if the Decision Diffie-Hellman assumption holds.

3.7.2 Re-encrypting metadata

Each user device maintains an integer counter called the epoch counter. It is initialized to zero, and represents the number of revocations that the user has performed. When a user device generates a new ABE key, Sieve automatically tags the key with an *epoch* attribute that is set to the current value of the epoch counter. The epoch attribute is a standard ABE attribute; until now, we have elided the *epoch* attribute in key descriptions, but we explicitly represent it in this section.

Suppose that a web service possesses the ABE key $K_{a_0, \dots, a_N, epoch=i}$, where i is a whole number. To remove the service's access permissions, the user first re-encrypts the affected data using homomorphic encryption. The user then increments the epoch counter to $i + 1$. Next, the user generates a new metadata block for each re-encrypted object, inserting the new k . The user encrypts the new metadata block and uploads it to the storage provider; the metadata is encrypted using the updated ABE key $K_{a_0, \dots, a_N, epoch=i+1}$. Finally, the user sends the new ABE key to any non-revoked web services who possess the old version of the key from epoch i ; remember that if the user gives multiple web services access to a_0, \dots, a_N , those services will receive the same ABE key.

Additional web services may require new ABE keys, depending on how the attributes in ABE keys overlap. For example, consider two web services: the first possesses $K_{(a_0 \text{ OR } a_1) \text{ AND } epoch=0}$, and the second has $K_{(a_1 \text{ OR } a_2) \text{ AND } epoch=0}$. Both keys grant access to a metadata block with attributes $a_1 \text{ AND } epoch = 0$. To revoke the first key, Sieve re-

encrypts the metadata block using the attributes a_1 AND $epoch = 1$. As a result, the second, non-revoked web service loses access to the block. Thus, Sieve must give an updated key $K_{(a_1 \text{ OR } a_2) \text{ AND } epoch=1}$ to the second service.

When Sieve re-encrypts a data object, the object's GUID stays the same, but its symmetric key changes. This invalidates any cached keys that are held by web services. So, when a service receives an updated ABE key, the service discards any cached symmetric keys that were decrypted using the old version of the ABE key. The signature is unaffected because the signature is on the cleartext data, which remains unmodified.

3.7.3 Additional details

A Sieve user will often possess multiple devices; for example, a single user might possess a smartphone, a tablet, a laptop, and a quantified life device like a Fitbit. If a user has multiple devices, then the device which initiates a revocation will broadcast the revoked key and the new epoch counter to the other devices. This ensures that the other devices do not use an old epoch number to encrypt new metadata blocks. Network partitions may delay the rate at which devices learn about the revocation, so when a device receives a revocation notice, it proactively rekeys any data and metadata that it mistakenly encrypted using the revoked key. Each revocation notice will have the issue time to help devices figure out what data was uploaded with the old epoch number. Computationally weak devices like Fitbits can delegate rekeying work to more powerful devices like laptops.

Revocation messages are signed by the public key of the device that issued the message. Devices learn about each other's public keys at Sieve initialization time, and later, when the user adds a new device. By signing revocation messages, Sieve prevents arbitrary devices from injecting fraudulent revocation notices.

To the best of our knowledge, Sieve is the first ABE system which supports full re-keying of *both* data and metadata. Prior ABE systems either cannot revoke keys at all [?], or can only revoke access to metadata [? ? ?]; in the latter case, data remains encrypted with revoked symmetric keys, leaving that data vulnerable to storage server compromise or negligence.

3.8 Protecting Against Device Loss

At initialization time, Sieve creates an ABE master secret. Sieve uses the master secret to derive the ABE decryption keys that are given to web services. Thus, the entire cryptosystem is compromised if the master secret is lost.

In a straightforward implementation of ABE, each user device has a copy of the master secret. However, portable devices like smartphones and tablets are often lost [?], meaning that a naïve implementation of ABE exposes the master secret to great risk. Even if users encrypt the master secret with a password-derived key [?], users often pick weak passwords [?], giving the master secret weak protection in practice if a device is lost.

To mitigate the impact of lost devices, Sieve uses Shamir secret sharing [?] to partition the master secret across a user's devices. In a (k, n) sharing scheme, the secret is divided across n devices, and k shares are required to reconstruct the secret. In the context of Sieve, this means that when a user device needs to generate an ABE key, the device must first gather $k - 1$ shares from other devices. Only then can the device assemble the master secret, generate the ABE decryption key, send the key to the web service, and then delete the local copy of the assembled master secret.

When the master secret is being assembled, Sieve requires the user to explicitly authorize each participating device to release its local share. By default, Sieve uses a k of 2, so this authorization scheme is similar to two-factor authentication [?]—a user cannot generate an ABE decryption key unless she controls two separate devices (e.g., a laptop and a smartphone). This means that, if an attacker finds a single lost device, that device cannot generate the master secret.

Sieve also employs secret sharing to protect the user's signature key. During uploads to the storage provider, the signature key is used to authenticate the client-side of the SSL/TLS session. Thus, the storage provider can reject fraudulent upload attempts from arbitrary devices.

Secret sharing protects the ABE master secret and the user's signing key. However, a lost device possesses a device key that is used to authenticate messages from that device. An attacker with a lost device can try to use the device key to subvert the revocation protocol

(§3.7). For example, if a malicious lost device can roll back the epoch to a *smaller* number, uncompromised devices will upload new data that can be decrypted with revoked keys. To prevent attacks like this, Sieve relies on the multi-factor authentication that is built into the secret sharing protocol—revocation requires a device to assemble the master secret, and assembling the master secret requires the user to possess multiple devices.

To add or remove devices from the secret sharing scheme, or to change k , the user must invalidate the old shares. To do so, the user must find k devices to participate in a new secret sharing exchange that uses the updated k and n .

Sieve provides no protections against a subverted device that the user believes is not lost or malfunctioning. For example, if the user wants to upload data from the smartphone that she is currently using, and the smartphone has a rootkit, the phone can arbitrarily delete the user’s data, upload garbage, or improperly revoke keys.

3.9 Minimizing ABE Overheads

Until now, we have assumed that clients perform two encryptions for every object upload: an ABE encryption for the metadata block, and a symmetric encryption for the data block. ABE is a public key cryptosystem, so ABE operations are much slower than symmetric ones. Fortunately, Sieve clients can use several techniques to reduce the frequency of ABE operations.

The simplest approach is for clients to store multiple objects inside each data block. Creating the associated metadata block will still require an ABE encryption, but subsequent writes and reads of the data block will only incur symmetric cryptography costs—clients can update the data block in-place, without changing the metadata, and third parties can cache the data block’s symmetric key to use during reads. For example, a smartphone with a GPS unit might use a single data block to store a month’s worth of location data. The phone appends new location samples to the current month’s data block, creating a new data block and metadata block when a new month begins.

Clients can also leverage more complex *storage-based data structures*. For example, as shown in Figure 3-2, a Sieve client can use indirect GUIDS like a Unix file system uses

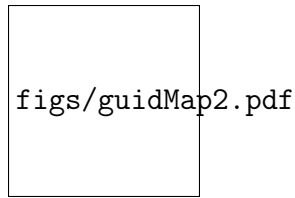


Figure 3-2: An example of a storage-based data structure. Using indirect GUIDS, the metadata block at the top points to a data block that only contains GUIDS. Those GUIDs point to raw data blocks. Raw data blocks can also embed pointers, as demonstrated by the simple tree structure that links the data blocks.

indirect data pointers. In this scheme, the top-level GUID for a storage-based data structure refers to a metadata block that points to a *GUID map*. The GUID map is just a data block that contains a symmetric key and additional GUIDs; those GUIDs point to raw data blocks that are encrypted with the symmetric key. Once again, clients and web services eliminate ABE costs by encrypting many objects with the same symmetric key, and caching that key.

Having many, smaller data blocks instead of fewer, larger data blocks is useful if the storage provider does not allow partial block writes (meaning that all writes force the client to upload at least a block's worth of data).¹ For similar reasons, multiple small blocks are useful if the symmetric cipher does not allow new data writes or updates to random offsets in the ciphertext.²

A raw data block can also embed GUIDs which reference other data blocks. This allows a client to build more complex data structures than flat arrays of data blocks. More complex structures may be preferable if they can reduce the number of wide-area data fetches that web services must perform. For example, Figure 3-2 shows a simple tree with a single parent and three children; by convention, the parent of the tree is the first entry in the GUID map. Each data block can hold multiple items, but when a block fills up, the client creates a new data block, adds the associated GUID to the GUID map, and then updates any internal GUIDs within preexisting data blocks. A third party whose ABE key decrypts the metadata block can traverse the tree structure without additional ABE operations, since all of the data blocks are encrypted with the same symmetric key.

¹For example, Amazon's S3 allows partial reads, but not partial writes [?].

²Most commonly used block cipher modes, such as CBC and CTR mode, do not easily support new writes to random offsets.

For future modifications to the data structure, such as adding new or updating existing data blocks or creating new pointers and GUIDs, the user caches the relevant GUIDs and keys (data and GUID map) allowing her to modify the contents of the data structure without creating a new metadata block, thus, avoiding an ABE operation. Similarly, a web service could and should cache the same information and use it to download any changes to the data structure to avoid incurring the cost of ABE operation. However, the caching is acceptable because the user can revoke the web service’s access to her un-accessed data using Sieve’s revocation protocol (§3.7)

Each storage-based data structure defines a Python API for adding and removing objects, as well as traversing the entire structure. Sieve clients and web services cache the metadata blocks for storage-based data structures, and use the Python APIs to interact with those structures. Thus, Sieve clients and web services are insulated from the low-level details of GUID maps (although both parties can access raw storage if desired, and if Sieve’s ABE policies allow such accesses).

Sieve’s revocation protocol is compatible with GUID maps, which are re-encrypted in place, just like any other data block. However, when Sieve determines that a metadata block must be re-keyed, Sieve checks whether the metadata refers to a GUID map. If so, Sieve must chase the GUIDs in the GUID map to identify the raw data blocks that must be re-keyed. The revocation protocol does not change the GUIDs that are associated with re-keyed data blocks, so embedded GUIDs inside data blocks remain valid after re-keying.

Each data block that is referenced by a particular GUID map is encrypted with the same k . However, Sieve uses counter-mode encryption [?], and employs a different counter for each block. Thus, if an attacker learns the cleartext for one ciphertext block, the attacker does not have an easier job of decrypting other ciphertext blocks with the same k .

3.10 Relabelling

In Sieve, a user may relabel an object. For example, a user can restrict access by adding an additional attribute to the object. A user can also remove attributes, or swap one attribute for another. Each web service’s ABE decryption key does not change unless the relabelling

results in an over-privileged web service, the user has to perform revocation and re-issue the web service's key to ensure that it has the correct access policies.

To implement relabelling, a user's Sieve client performs three actions. First, the client replaces the old metadata block on the storage server with a new one that contains a new symmetric key and is ABE-encrypted using the new attributes. Second, the client uses homomorphic encryption to re-encrypt the object under the new symmetric key on the storage server. Finally, the client updates storage-based references to the object, ensuring that the references adhere to the object's new access policy. The client can locate these references because the client knows the old attributes for the object, the new attributes for the object, and the attributes for all of the user's metadata blocks. Thus, the client can determine which references must be patched. For example, suppose that a user has two storage-based data structures S_0 and S_1 ; further suppose that, due to relabelling, an object must move from S_0 to S_1 . By inspecting the object's old attributes, the client determines that the object was originally referenced by S_0 . The client homomorphically re-encrypts the object using symmetric key k' , traverses S_0 to remove any references to the object, and then adds a $\langle GUID, k' \rangle$ reference for the object to S_1 . Sieve performs the traversals, removals, and additions using the APIs defined by the storage-based data structures.

3.11 Discussion

3.11.1 Alternative policy languages

Attribute-based disclosure policies are easy for users to understand, and these policies naturally map to ABE cryptosystems. However, ABE cannot express arbitrarily complex policy functions. Garbled circuits [?] and functional encryption [?] are Turing complete, but they are prohibitively slow. For example, garbled circuits decrypt AES data at a rate that is four orders of magnitude slower than native AES decryption [?]. Relative to functional encryption and garbled circuits, ABE is several orders of magnitude faster. We believe this trade-off is worthwhile.

3.11.2 Paying for storage

In Sieve, each user places her objects in private cloud storage. Someone must pay for that storage. One option is for ad networks to pay. In Sieve, ad networks can be third parties, and they can receive ABE keys to access user data. Using a micropayment system like FileTeller [?], advertisers could pay for the right to collect longitudinal data about a user, and generate targeted advertisements based on that data. By deferring user storage costs, advertisers would encourage users to continue to declassify a subset of their data. Indeed, since each user now stores all of her data in a single place instead of multiple locations, ad networks would gain access to more contextual information than in the web ecosystem, even if users choose which objects to reveal [?]. Thus, Sieve might enable a happy middle ground in which users gain explicit control over the data seen by third parties, and third parties willingly subsidize private user storage in return for better contextual information.

If ad-driven storage subsidies are poorly designed, they may lead to perverse trade-offs between subsidy amounts and the required levels of data disclosure. A full study of such interactions is beyond the scope of this paper. For now, we merely observe that some users may opt out of the subsidy system entirely. These users will have to pay for their own storage, but there is reason to believe that they would do so. Well-known sites like Pandora, Slashdot, and OkCupid already allow users to pay a small monthly fee to remove advertisements, so there is a preexisting demographic which is willing to pay money in exchange for better privacy. The popularity of open source applications also demonstrates that developers are willing to make high quality applications without the expectation of direct payments from users. Thus, we believe that Sieve’s application model is realistic.

3.11.3 Efficient data importing

In the current web ecosystem, users explicitly submit data to web services, making it easy for those services to determine when new information has been created. In Sieve, users submit new data to the storage provider. However, user devices know the tags that are associated with both new data items and web service ABE keys; thus, when a device uploads an object of interest to a particular service, the device can proactively notify the service of the upload.

Storage-based data structures (§3.9) also make it easy for services to identify new data. For example, using a storage-based log, user devices can append new data to the head of the log. A service can cache the GUID and the symmetric key for the log head, and periodically check the beginning of the log for new objects.

3.11.4 Anonymity across services

Some users may not want to be tracked across different web services. For example, a user might be comfortable sharing data with services *X* and *Y*, but uncomfortable with *X* knowing how she interacts with *Y*, and vice versa. Sieve cannot restrict what services do once they possess user data, so Sieve cannot prevent *X* and *Y* from pooling their data and trying to correlate user behavior across both services. Users can employ various techniques to make tracking more difficult. For example, proxies like Tor [?] allow users to hide their IP addresses from web services. However, users can also establish a unique login identity for each web service, or lobby web services to use anonymous credential systems [?]. Unfortunately, Tor and anonymous credential systems rely on network proxies that hurt application responsiveness, and seemingly anonymized data sets can still reveal sensitive user information to machine learning algorithms [?]. Thus, providing anonymity on the web is still an important area for future research.

Chapter 4

Implementation

Our Sieve prototype consists of a Sieve client, a storage provider daemon, and a Sieve import daemon that is run by third parties. Each component is written in Python, and uses PyCrypto [?] to implement RSA and AES. For ABE operations, we use the libfenc [?] library with elliptic curves [?] from the Stanford Pairing-Based Cryptography library [?]. To build Sieve’s key homomorphic symmetric cipher [?], we use the Ed448-Goldilocks elliptic curve library [?].

The storage provider daemon uses BerkeleyDB [?] to store encrypted data blocks, and MongoDB [?] to store metadata blocks. For each data block, the key is a GUID, and the value is a symmetrically encrypted object. For a metadata block, the key is a set of cleartext ABE attributes, and the value is an ABE-encrypted GUID and symmetric key. Metadata blocks are indexed by their attribute fields, and all metadata blocks for a particular user are stored in a MongoDB collection.

The JavaScript code in a web site interacts with the local Sieve client using a small RPC library that we provide. When a web site initially requests access to a user’s data, the site’s JavaScript sends an XMLHttpRequest to a localhost webserver run by the Sieve client. The Sieve client then displays a GUI that allows the user to define an access policy for the site, and send the associated ABE key to the site’s web server.

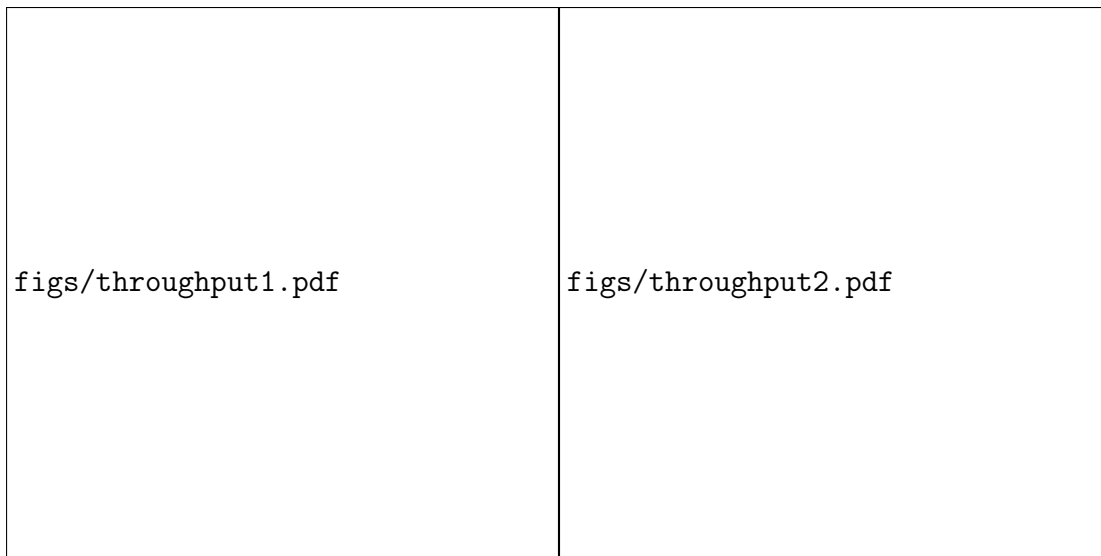
Chapter 5

Evaluation

In this section, we explore a single high-level question: is Sieve practical? By practical, we mean two things: is it easy to integrate Sieve into existing applications, and can applications achieve good performance when using Sieve? To answer this question, we integrated Sieve with two applications. The first was Open mHealth [?], an open-source web service that allows users to analyze their health data. We also integrated Sieve with Piwigo [?], an open-source online photo manager. We show that the integration was straightforward, and that the end-to-end application pipeline can handle realistic workloads.

All experiments ran on a machine with 2.4 GHz 10-core Intel Xeon E7-8870 CPUs and 256 GB of RAM. We ran each experiment 50 times, and we report the average (standard deviations were small). Sieve used 2048 bit RSA with SHA256 to sign user objects. ABE operations used 224-bit MNT curves [?]. To symmetrically encrypt those objects, Sieve used 128-bit AES in CTR mode, or Ed448-Goldilocks elliptic curves in randomized counter mode. The latter cipher is key homomorphic, but the former is not; by comparing Sieve’s performance with these ciphers, we could measure the cost of supporting key revocation (§3.7). All web servers ran on the test machine’s loopback interface, to minimize network latency and focus on Sieve’s cryptographic overheads.

All GUIDs were 64 bits long. Thus, a metadata block which contained a GUID and an AES key was 24 bytes in size, whereas a metadata block which contained a GUID and an Ed448 key was 64 bytes long.



(a) Encryption speed: ABE and Ed448 in randomized counter mode. (b) Encryption speed: ABE and AES in CTR mode.

Figure 5-1: Encryption throughputs for Sieve, as a function of 1) the size of the data to symmetrically encrypt, 2) the percentage of symmetric data encryptions which also require the ABE encryption of a metadata block, and 3) whether the cipher is AES or key homomorphic Ed448. All experiments assume that each metadata block has five attributes, and each ABE key has 10. Performance trends for decryption are similar.

5.1 Case Studies

We present the results of our integrations with Open mHealth and Piwigo, and then discuss these results.

5.1.1 Open mHealth

Open mHealth allows users to upload medical data to a web server that will analyze the data and provide explanatory visualizations. To integrate Sieve with Open mHealth, we first modified the Open mHealth client to upload data via the Sieve client instead of directly to the Open mHealth server. We then ran a Sieve import daemon on the Open mHealth web server, configuring the daemon with the data schema that the Open mHealth analytics engine uses. These modifications required approximately 200 lines of code to be changed in the Open mHealth platform.

To test the end-to-end performance of the application pipeline, we used Open mHealth’s data generator to create a week’s worth of health data. The data included information like weight, blood pressure, physical activity, and heart rate. Each day had approximately 14 data points. For each data point, the Sieve client added attributes like the date that the sample was collected, the name of the associated user, and the type of data represented by the sample. The Sieve client used a single storage-based data structure to store the samples for an entire week.

The cost for the user to upload the first data point at the beginning of a week was 0.56 seconds; the cost was dominated by ABE encryption. Uploading subsequent data points proceeded at the throughput of the symmetric cipher, requiring 17.1 ms per data point for AES, and 38.5 ms for Ed448.

The mHealth server used the Sieve import daemon to download user data. If the mHealth server had no cached GUIDs or symmetric keys, then importing a week of data required 1.1 seconds. In this scenario, the server had to download the metadata block, decrypt it with ABE, download the raw data block, and then decrypt that block using a symmetric cipher. If the server possessed cached GUIDs and symmetric keys, then importing a week of data took only 0.095 seconds with AES, and 0.75 seconds with Ed448.

5.1.2 Piwigo

The standard Piwigo client allows users to upload photos from local storage to the Piwigo web service. We modified the client to upload data to a Sieve storage provider, and we modified the server-side Piwigo code to fetch user data via the Sieve import daemon. These modifications required approximately 250 lines of new Piwigo code.

To test the end-to-end performance, we uploaded a 375 KB photo which had three tags (location, date, and username). If the Piwigo client used AES as the symmetric cipher, the upload required 0.57 seconds if a new, ABE-encrypted metadata block also had to be generated. If the client used a storage-based list to avoid the creation of a new metadata block, the upload cost was only 0.06 seconds.

As we explain in more detail in Section 5.2, current Ed448 implementations are slower and less optimized than equivalent AES implementations. Thus, when applications use Ed448, the upload time for a large object is dominated by Ed448 encryption costs, regardless of whether ABE costs are incurred. If the Piwigo client used Ed448, the upload cost was 6.1 seconds if the client also had to generate a new metadata block. By using storage-based data structures to avoid ABE operations, the upload cost dropped to 4.2 seconds. Note that, from the user’s perspective, *uploads are asynchronous*. Thus, multi-second upload times are not in the critical path of user-facing activities.

Download times for the Piwigo server demonstrated similar trends. With cached GUIDs and symmetric keys, downloading a photo required 0.14 seconds using AES, and 5.9 seconds using Ed448. Without cached metadata, a download required 0.44 seconds with AES, and 6.3 seconds with Ed448.

5.1.3 Server-side per-core throughput

Our implementation of the storage daemon uses BerkeleyDB to store data objects. The daemon logic is simple, meaning that the daemon can import data at the raw speed of the BerkeleyDB write path. For Open mHealth, the write speed was roughly 50 MB/s per server core, which represented 16,500 users uploading a week’s worth of data every second. For Piwigo, the write speed was roughly 200 MB/s per core, corresponding to 550 photo uploads

per second (assuming a photo size of 375 KB). Write throughput was better for Piwigo due to BerkleyDB handling large writes faster than small ones.

We also tested per-core throughput for the import daemon. For Open mHealth using AES, a single core could download and decrypt a week's worth of data for 420 users in one minute; with Ed448, a core could import 70 users' data in one minute. Given a photo size of 375 KB, Piwigo was able to import 235 AES-encrypted photos or 14 Ed448-encrypted photos in one minute. In all experiments, 20% of object imports required the download and ABE-decryption of a metadata block. We believe that 20% is high, since an arbitrary number of objects can be referenced by a single metadata block.

5.2 Microbenchmarks

5.2.1 Encryption speed

Sieve requires clients to symmetrically encrypt each data object before uploading it. Some fraction of uploads will also require clients to ABE-encrypt a metadata block. Reducing that fraction is important, because ABE operations are much slower than symmetric operations.

Figure 5-1 quantifies the performance differences between ABE and the symmetric ciphers. For 10 KB objects, pure ABE encryption throughput is 1.1 KB/s, whereas pure Ed448 throughput is 23.8 KB/s and pure AES throughput is 43.5 KB/s. Although clients can perform data uploads asynchronously, in the background, the computational costs for ABE are still quite high. Thus, hybrid encryption (§3.5) and the optimizations from Section 3.9 are crucial for minimizing the number of ABE operations.

Another observation is that the throughput for larger objects is higher than the throughput for smaller objects because there is a fixed cost for an ABE operation regardless of the size of the object. Encrypting larger objects does not take that much longer than smaller objects because the ABE operations dominate the encryption time.

For 1 MB objects, the performance gap between AES and Ed448 grows—AES throughput is 12 MB/s, and Ed448 throughput is only 120 KB/s. However, Ed448 is a new elliptic curve, with immature implementations relative to AES. We expect Ed448's performance to

Operation	Time (s)
Generating 10 attribute key	0.46
Generating 20 attribute key	0.64
Re-encrypting a metadata block (10 attributes)	0.63
Re-encrypting a metadata block (20 attributes)	0.91
Re-key 100 KB data block	0.41

Figure 5-2: Computational overheads for key generation and revocation.

improve as its implementations receive more optimization effort.

5.2.2 Key generation and revocation

Figure 5-2 describes the costs that Sieve pays for generating new ABE keys, re-encrypting metadata blocks, and re-keying a 100 KB data block. The creation of new ABE keys is rare, and only occurs when a new service requests access permissions, or an old service receives modified permissions (possibly as the result of an epoch number increasing after a revocation (§3.7)). During revocation, the metadata blocks associated with the revoked ABE key must be re-encrypted; however, those metadata blocks will typically point to a much larger number of raw data blocks (§3.9), so the overall re-encryption cost of revocation is governed by the speed with which raw data can be re-keyed.

5.2.3 Attribute matching

When the storage provider receives a data access request from a third party, the storage provider must locate the metadata blocks whose attributes match those of the access request. Sieve makes the matching process fast by storing metadata blocks in a database and indexing those blocks using their attributes.

However, the results are unsurprising, since modern databases are good at building indices. For example, in one experiment, we injected 1 million metadata blocks into MongoDB; each metadata block had 10 randomly selected attributes from a universe of 35 possible attributes. Then, we submitted access queries in which each query contained 5 random attributes joined with a random set of ANDs and ORs. Each query took 0.13 ms to

complete.

5.2.4 Secret-sharing

Sieve partitions the ABE master key and the RSA signing key across multiple devices, ensuring that a lost or stolen device will not store a full copy of sensitive cryptographic information. The secret sharing protocol is cheap: ignoring network latencies, and assuming that $k = 2$ and $n = 5$, splitting a 2048 bit object like an RSA key requires 0.04 ms, and reconstructing that key requires 0.09 ms.

Chapter 6

Related Work

Sieve is related to prior work in several areas: using cryptography to protect data on untrusted servers; using ABE to protect storage; using oblivious RAM; and other delegation protocols. The rest of this chapter discusses these areas of related work in turn.

6.1 Untrusted servers

Browser extensions like ShadowCrypt [?] transparently encrypt the data that a browser sends to unmodified cloud servers. Intentionally encrypted cloud stores like SUNDR [?], Depot [?], and SPORC [?] provide stronger consistency semantics in the face of server-side misbehavior; application logic runs solely on the client-side, over cleartext data, with clients exchanging encrypted data with servers. Other systems that store encrypted data on servers and run application logic on the client-side include BStore [?] and DepSky [?]. All of these systems prevent data leakage due to server compromise or malice, but they are incompatible with applications that leverage server-side computation to add value to raw user data. In contrast, Sieve is totally compatible with server-side computation.

In CryptDB [?], a web application consists of clients, an application server, and a back-end database. The database only contains encrypted data. Using SQL-aware encryption, the application server can execute queries over the encrypted data without revealing cleartext to the database. However, the application server does see cleartext, and can leak user data if compromised. Mylar [?] eliminates the need for an application server, but

restricts the encrypted server-side computation to keyword searches. In both CryptDB and Mylar, applications control how user data is shared. In Sieve, user data is decoupled from applications, and users can selectively disclose individual objects to third parties.

Privly [?] allows users to upload encrypted data to a storage server, and share hyperlinks to that data. The hyperlinks can be embedded in sites like a Facebook page, but the hyperlinks reveal no cleartext to the owner of the embedding site. Users register their decryption keys with Privly's browser extension. Later, when the user visits a page and her extension finds a Privly hyperlink, the extension transparently fetches the encrypted data, decrypts it, and rewrites the page's HTML, replacing the Privly link with the cleartext data. Privly does not support the server-side computation enabled by Sieve.

6.2 ABE-protected storage

Persona [?], Priv.io [?], and Cachet [?] use ABE to selectively expose encrypted user data. In Persona and Priv.io, each user keeps her data in private cloud storage; in Cachet, data is stored in a peer-to-peer, distributed hash table. Unlike Sieve, these systems cannot delegate access to third party services, breaking compatibility with rich applications that require server-side computation. Persona, Priv.io, and Cachet also trust each device for the lifetime of the system, whereas Sieve distributes this trust. Finally, Sieve provides a concrete revocation protocol that safeguards user data if storage servers are compromised. Priv.io has no revocation strategy, and Persona suggests re-keying data, but does not provide a specific protocol. Cachet does implement revocation, but requires a trusted proxy which must interpose on all decryption operations, even in the common case that revocation is not underway [?]. Cachet's revocation scheme also does not re-encrypt data on storage providers, leaving data encrypted under revoked keys vulnerable to subsequent compromises of the storage provider.

6.3 Predicate encrypted storage

GORAM [?] is an oblivious RAM system that allows users to selectively share their cloud data with other users. Clients place encrypted data on servers so that servers cannot inspect it, and the clients' access patterns are hidden from servers using ORAM shuffling techniques [?]. Like Sieve, GORAM tags data objects with attributes; unlike Sieve, GORAM uses attribute-hiding predicate encryption [? ?] to restrict knowledge of attributes to the clients that possess matching decryption keys.

GORAM's use of oblivious RAM and predicate encryption provides stronger security than Sieve, but at a performance cost. To hide data access patterns from storage servers, GORAM clients must perform $O(\text{polylog}(n))$ more data accesses. Hiding attribute values using predicate encryption substantially increases GORAM's ciphertext size and its encryption and decryption times. In comparison, Sieve provides lower-latency web services with less user friction. GORAM also forces users to determine a priori the maximum number of principals that can be mentioned in access control lists; if this list changes, a user must re-initialize her database. GORAM has no revocation scheme and assumes that users have one trusted device whereas Sieve provides a complete revocation scheme and prevents against device losses that might lead to a compromise of the whole system.

6.4 Access delegation schemes

OAuth [?] is a widely-used protocol for sharing cloud data across different web services. OAuth policies are written by web services, not by users, so users lack true authority over their access controls. OAuth also does not leverage cryptography to protect user storage or enforce access policies. As a result, users have no strong assurances about how their data is exposed. OAuth is also vulnerable to various kinds of data leaks [? ?]. AAuth [?] is an extension of OAuth which uses cryptography to delegate access to encrypted data, but relies on the existence of various trusted parties to enforce access policies. In Sieve, users generate their own policies and distrust the storage server and third party applications. Sieve's protocols are simpler, and Sieve's policy language is richer than AAuth's fixed policy

schemas. OAuth's revocation scheme, like Persona's, only revokes access to the metadata block, but consider cached symmetric keys that might leak cleartext data when the cloud provider is compromised.

The OAuth protocol generates a token that principals use to access sensitive data. Web services define many other types of "bearer tokens." HTTP cookies [?] are a classic example. Macaroons [?] improve upon cookies, using chained HMACs to verify and attenuate capabilities as a macaroon is passed between multiple parties. Cookies and macaroons vouch for a principal's post-authorization status, whereas Sieve deals with the authorization itself.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Sieve is a new access control system that allows users to selectively expose their private cloud data to third party web services. Sieve leverages attribute-based encryption to translate human-understandable access policies into cryptographically enforceable restrictions, and is the first ABE system to protect against device loss and supports full revocation of both data and metadata. Unlike prior solutions for encrypted storage, Sieve is compatible with rich, legacy web applications that require server-side computation. As a proof of concept, we integrated Sieve with an open-source health monitoring service, showing that Sieve is a practical mechanism for protecting user data.

7.2 Future Work

There are a number of interesting directions to follow up on Sieve. Sieve provides an access control mechanism for user-managed data for web applications. An interesting question is how can users interact with each other in a peer-to-peer setting. For example, can users share data or do computations on each other's data revealing minimal data? Another interesting question is whether users can participate in surveys without revealing to the web application their value. For instance, assume Amazon wants to know how many people purchased an item. Can users release their data such that Amazon only learns the sum but not any

individual answers.

Finally, another area of interesting work involves Sieve user data after it has been released to web applications. Is there a way to control the flow of information out of applications based on ABE-policies? Is there a way to extend Sieve to regulate the export of data between applications and maybe other users.

Sieve has envisioned a new web ecosystem that is beneficial for both applications and users. It is important to see how Sieve can be applied to other important web applications.

Bibliography