CRYSTOPOR

A Platform for Secure Storage

by

Alexandru Totolici

B. Science, The University of British Columbia, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

June 2013

© Alexandru Totolici, 2013
Abstract

Cloud computing continues to grow as a preferred development and deployment model, enabling new types of applications to be built while keeping resource requirements convenient for developers. Sacrifices have been made, however, in terms of the security and protection of data stored in these cloud applications, leaving users vulnerable to information leaks and invasions of privacy.

We present CRYPSTOR, a storage platform design that addresses data security in the cloud computing context through the use of encryption, while maintaining the desirable properties of the current model, such as efficient storage and sharing.
Preface

I am the contributor of all the thesis work, under the supervision of my two supervisors, Bill Aiello and Andrew Warfield.
# Table of Contents

Abstract ................................................................. ii

Preface ................................................................. iii

Table of Contents .................................................. iv

List of Figures ......................................................... vi

Glossary ................................................................. vii

Acknowledgements ................................................... ix

1 Introduction ......................................................... 1

2 Related Work ....................................................... 4

3 Design and Architecture ......................................... 7
   3.1 Threat Model ................................................... 7
   3.2 Application Models .......................................... 9
      3.2.1 By Infrastructure .................................... 9
      3.2.2 By Level of Centralization ......................... 10
   3.3 Design Goals .................................................. 12

4 Cryptographic Engineering ...................................... 13
   4.1 Building Blocks ............................................. 13
   4.2 Constructs ................................................... 14
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>System</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>5.1 Server</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>5.2 Client</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>5.2.1 Monitoring</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>5.2.2 Cryptography</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>5.2.3 Transport</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>Evaluation</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>6.1 Security</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>6.2 Performance</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>6.3 Usability</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Future Work</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>7.1 Security Hardening</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>7.2 Performance</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>7.3 Replacing Cryptographic Components</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>7.4 Privacy</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>7.5 Group Shares</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>7.6 Removing Centralization Requirements</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>28</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>29</td>
</tr>
</tbody>
</table>
List of Figures

Figure 4.1 Key generation for a new user ............... 15
Figure 4.2 Key generation for groups .................. 16
Figure 5.1 Client-side workflow ....................... 19
Figure 6.1 Overheads of the cryptographic operations ... 23
Glossary

AES-256  Advanced Encryption Standard, 256-bit key

API  application programming interface

GCM  Galois/Counter Mode, an authenticated cryptographic mode of operation

DAAS  Data(base)-as-a-Service

EC2  Amazon Elastic Compute Cloud

HMAC  Hash-based Message Authentication Code, SHA-512

IPC  inter-process communication

LAFS  Tahoe, The Least-Authority Filesystem

KDF  key derivation function

NIST  National Institute of Standards and Technology

OAEP  Optimal Asymmetric Encryption Padding

P2P  Peer-to-Peer

PBKDF2  Password-Based Key Derivation Function 2

PKCS  Public Key Cryptosystem

PKI  Public Key Infrastructure
**SPRNG**  Secure Pseudo-Random Number Generator

**RSA**  Rivest, Shamir, & Adleman encryption

**RSAES-OAEP**  RSA Encryption Scheme with Optimal Asymmetric Encryption Padding

**S3**  Amazon Simple Storage Service

**SAAS**  Storage-as-a-Service

**SHA-384**  Secure Hash Function, 384-bit digest

**TLS**  Transport Layer Security

**XaaS**  X-as-a-Service, a cloud component provider
Acknowledgements

Any list attempting to thank one’s mentors, influencers, and supporters is necessarily incomplete. None of us is as knowledgeable as all of us, and this author has sought advice and instruction from diverse and numerous sources. Yet, any version of said list would begin acknowledging my two generous supervisors, Bill Aiello and Andrew Warfield, to which I owe a great debt of gratitude for the chance they took on me, and the opportunities that stemmed from it. I am also incredibly thankful to Norm Hutchinson for being the second reader of this work and for his great feedback.

Members of the Networks, Systems, and Security laboratory have been an endless source of ideas and astute, pointed critique; one could not have hoped for a more intellectually-diverse and technologically-adept cohort to spend this time with.

My mother and father deserve my endless gratitude for the risks they took, undauntedly.
Chapter 1

Introduction

*The universe believes in encryption* — Julian Assange (2012)

The novelty of cloud computing as a mode of providing services to users has passed at this point, with the paradigm firmly entrenched in modern software engineering practise. The always-on nature of centralized resources, coupled with an increase in the number of devices users want to use in their interactions with these various services, have combined as the factors leading to the prevalence of this development and deployment pattern. The evolution of software applications beyond the end-user’s computing devices and into third-party hosted environments has enabled significant innovation in online services, now that developers can rely on the ubiquitous availability of servers. Where costs and logistics may have previously killed the growth of services such as Netflix [3], offloading infrastructural concerns to dedicated providers removes a lot of the barriers to entry that have traditionally made the development of ideas cumbersome and risky. Where previously a company may have needed to lease a data centre and purchase hardware for it, and would not have been able to grow it fast enough to keep up with demand if their service offerings turned out to be very popular with customers, nowadays only a credit card is needed to harness the vast resources of a service like Amazon *Elastic Compute Cloud (EC2)* or Microsoft *Azure*.
Despite the model’s lack of freshness (as evidenced by the plethora of usage examples), there is an alarming degree of immaturity—or, at worst, incompetence—on the part of developers when it comes to the security primitives employed for the benefit of protecting user data. Past security incidents experienced by cloud service providers could have had significantly reduced compromises of user data had adequate means of securing it been put in place. It is unsettlingly easy to have the information these services store for their users exposed to everyone, whether accidentally (e.g., due to a security compromise [15], or a software bug [2]) or intentionally and with the aid of the application provider (e.g., to comply with law-enforcement requests [50]). The failure to adequately secure this information with mechanisms that would permit only the user to access it is not technical, but rather—if we are to maintain an optimistic point-of-view—one of developer convenience. Strong protections can be built in, by design, and do not require the loss of functionality as far as the user is concerned (although, as we discuss, some degradation in performance and quality may occur).

It is now possible for a software engineer to write and deploy a cloud-based application entirely using services provided by third-parties: back-end storage [5] and databases [9, 18, 19], message queues [10, 16] and task pipelines [17], notifications [7], and even front-end hosting [21]—all without managing a single server directly.

The ease of doing all of this just by calling into a service provider’s application programming interfaces (APIs) hides the complexities of the underlying mechanisms, and often developers lack the ability to understand or audit the systems they are asked to trust. These conveniences have enabled sloppy development hygiene, and a promiscuity of component pairings: anything goes, just so the service works. While the economic interests of provider and developer are well-aligned and should ensure no malicious behaviour on the part of either, the inherent increase in system complexity resulting from the involvement of additional “moving parts” (i.e., the service provider), translates to an increased overall attack surface. And while having a provider’s dedicated team work around the clock on
securing the particular service offered to developers is reassuring, central-
ization of resources makes the providers high-impact targets for attackers.

The security of the stored data seems to be treated as a second-tier fea-
ture, one not among the myriad of advertised conveniences otherwise be-
stowed upon developers. Sure, stored data is partitioned and isolated from
other tenants, replicated on redundant servers across multiple, geographically-
diverse data centres, and transferred across encrypted tunnels via Transport
Layer Security (TLS) or equivalent. But these are minimal requirements for
data integrity (and in the case of TLS, avoidance of trivial attacks); when at
rest, the data is not encrypted, and thus completely exposed to the whims
of any of the aforementioned attackers.

Application developers have, of course, the ability to roll out their own
security layers before sending bytes to a storage provider’s back-end; in
fact, some popular services [12] do exactly that before using Amazon Sim-
ple Storage Service (S3) [5]. They also maintain the encryption and decryption
keys, therefore merely shifting the risk from storage provider onto
themselves, leaving the end-user just as vulnerable. We visit the proper-
ties of the most notable such systems in Chapter 2 and observe that exist-
ing platforms stop short of providing the kind of user data protection we
consider adequate. We already expect the storage layer to provide certain
guarantees with respect to data integrity and availability, why not confi-
dentiality as well?

In the belief that a system embedding the necessary security primitives
in the storage framework’s API would spur developer adoption through
convenience, we present a design for object storage in Chapter 3 that ad-
dresses our goals. To demonstrate the viability of the proposed system,
we implement a prototype service for a popular data access pattern—users
keeping files synchronized across devices and sharing them—in Chapter 5
Chapter 2

Related Work

We emphasize that encrypted storage itself is not a contribution of this work, and in fact prior systems have touched on many, but not all, of the properties we desire for our system.

Tahoe (LAFS) [51] is “a storage grid designed to provide secure, long-term storage”, using cryptography and erasure coding to provide security and integrity guarantees for the data it stores. It is a capability-based storage system [35], meaning access to an object is granted by sharing an unique identifier pointing to it (which is both necessary and sufficient). LAFS supports both immutable and mutable objects, and enforces different privileges to those files in the form of verify, read, and write capabilities. LAFS is targeted as an infrastructure component, and indeed as a storage grid it would not be trivial to deploy by application providers. It is, however, the most practical such system currently in use.

S3 provides an API-level facility for clients wishing to encrypt prior to upload [6], using envelope encryption. Plaintext is encrypted under a symmetric key, which is in turn encrypted with a client envelope key (i.e., password). Both cyphertext and encrypted envelope are stored on the storage provider’s server. In case the client’s key is compromised, the symmetric data key does not need to be changed, only the envelope re-created with a new client key.

SUNDR [44] and Depot [45] are both network file systems that address
issues of integrity, accessibility, and authorized access over the stored data. These added protections are meant to detect incorrect or accidental changes to the data, and reduce the level of trust in the provider, but not protect from disclosure or leaks; for that, higher layers are made responsible, i.e., the application.

SPORC [38] removes the need for trust in the server, presenting it only with encrypted information. The role of the central node is to globally-order operations, and clients are able to detect if the server is misbehaving in that regard. SPORC also encrypts operations on the data, so the server is unable to know anything about what clients are doing.

SUNDR, Depot and SPORC all use variants of fork consistency (originally introduced by the authors of SUNDR [46]) to ensure that misbehaving servers would have to maintain alternate histories and present those to clients. All inter-client operations would fail, however, because the histories would not be reconcilable, which leads to quick detection of all errors, be they intentional or not. SUNDR and SPORC also use operational transformation [37] as the mechanism for data manipulation, by storing full objects and the set of operations that were performed on them by clients; in this manner, space and bandwidth can be saved by avoiding large uploads for small changes in the underlying data, and conflicts can be more easily represented.

CryptDB [49] is an encrypted database preventing the administrators and the Data(base)-as-a-Service (DaaS) provider from being able to observe its contents or queries. Records are each layered in various cryptographic functions in order to support a wide range of operations. When needed, CryptDB ‘peels the layers’ to the level required for the correct execution of queries, thus maintaining the highest possible level of data confidentiality at rest. This approach can be extended to other centralized services in order to enable certain kinds of functionality in an application (e.g., search or spam detection) without sacrificing security by exposing the entire plaintext to the provider.

\[1\] Policies can specify a minimum layer beyond which no decryption will happen.
Commercial solutions in this space either encrypt data as it moves from the enterprise to the cloud [8, 11, 24, 28, 31], or attempt to simplify the deployment and management of encrypted storage in cloud-based virtual machine environments, such as EC2 [23, 26]. These solve part of the problem, but end-users must still trust the provider because that is where key management takes place and where policies are enforced.
Chapter 3

Design and Architecture

As we noted previously, cloud-based applications are subjected to various possible combinations of their components, eschewing traditional client-server models by the inclusion of third-party X-as-a-Service (XaaS) providers. For the purposes of this system, we denote three principals: the Storage-as-a-Service (SaaS) provider where data is ultimately kept, the application itself, and its clients (i.e., end-users). We consider other XaaS providers that may be involved to have the same privileges with regards to the data as the application does, because ultimately it is the software that provides the interconnect between these components.

In approaching the design of our system, we must identify the infrastructural level at which we expect it to be deployed, as well as the service architectures we may be able to support, based on their level of centralizations. To further inform these key design features we must first formalize our threat model.

3.1 Threat Model

CYPSTOR makes the following security assumptions about the operational environment:

**Storage Servers**

Operators are providing a commodity and have a financial interest in
maintaining a good working relation with their clients. Either due to coercive force or security incidents, they may act maliciously toward clients and attempt to gather information on the data stored. Blocking inter-client communication altogether is undesirable, because it will lead to loss of customers. We think of providers as being ‘honest but curious’.

Information Leaks

Object data should not leak, which is the primary reason we employ cryptography throughout the system, client-side.

Information may leak about the popularity of certain objects, as the server sees the read requests issued for them. It is possible to create dummy requests in order to counteract this analysis, but the cost in terms of bandwidth may not be desirable.

There should not be any information leaked regarding either the object’s metadata, or the kinds of operations applied to the object. However, as this is an object store, relative sizes of the objects will be leaked; if files are broken up in blocks and dummy requests are generated in order to mask access patterns, size information can be hidden from the server.

Clients

Clients have a secure way of storing their encryption keys. In a decentralized model, particularly, this may require making certain cryptographic material semi-public, in order to allow users to easily bootstrap when they want to use a new application with their storage or log in from a new computer.

Clients establish relationships when they want to share data, e.g., by exchanging their public keys in order to generate envelope keys with which to encrypt and share objects. These links may be partially leaked to the server, though clients may request objects even without knowledge of the corresponding decryption key, which would confuse such analysis.
Application Code Integrity

Applications are safe and trusted, and will not broadcast clients’ encryption keys, sniff their passwords, or otherwise act maliciously. In practice this requires cryptographically signed applications, delivered via secure mechanisms such as HTTPS.

3.2 Application Models

3.2.1 By Infrastructure

We first recognize the major locations in the data flow path where encryption could take place, given an arbitrary application using cloud-based storage:

- **Back-end provider**, enabled in the cloud management infrastructure (e.g., Eucalyptus [14], OpenStack [20]);

- **Application developer**, as a shim layer interposed in front of the back-end storage;

- **Client-side**, by means of a library that pushes the execution of secure operations to the clients themselves.

These locales each have their own shortcomings with respect to our goals. Infrastructural changes are difficult to deploy because of the operational risk-adversity that is typical of providers. Having security-related mechanisms execute for all of their tenants may also make providers antagonistic, as the increase in computational requirements carries a significant financial cost. As an interposition layer that an application developer would deploy between their code and the storage back-end (in a nearly transparent manner), trust still remains with the developer, and while we assume the best of intentions on their part, programming mistakes can easily and completely undermine any protections put in place [2].
Finally, pushing more computations on the client-side may lead to performance penalties, particularly on mobile devices, and reduce the overall appeal of some applications. It is only here, however, that our goals can be successfully achieved, as complete trust in either back-end storage provider or application developer is not required.\footnote{This is not to say we assume either is malicious, because if that was the case, it is unclear why we are willing to run their code at all; rather, our system aims to avoid information disclosures the client sees as unintentional, regardless of whether they were to happen through a security breach or a policy claim.} If the security primitives live with the end-users, then providers can avoid the added computational costs of securing the service, and most importantly there are significantly lower risks of a massive data leak.

Of course, the application provider can be its own client, in effect using our system as a shim in front of the storage back-end, much like some incumbents already do (with custom, in-house solutions); we hope the ease of client-side deployment will remove most such use in practise, outside of those applications that are very hard to conceptualize without the provider having access to the raw form of the information\footnote{Banking is one such application, where in fact the provider has the data to begin with; but many other kinds of applications, including social networks, do not fall in this category as far as user interests are concerned.} We shall refer to ways in which developers can benefit from using CRYPSTOR as an interposition layer as well, when relevant.

### 3.2.2 By Level of Centralization

Client-side applications fall into one of a handful of operational patterns, dictated by their particular mix of centralized and distributed components. On one end of the spectrum, we have what are now typical cloud-based services, fully centralized with the application provider; on the other, we have complete distribution through end-users or federating intermediaries. CRYPSTOR aims to be able to satisfy the needs of applications spanning this entire spectrum. The application’s architectural decisions inform the approach for the back-end, which is why it falls with CRYPSTOR to satisfy the breadth of patterns in this respect. A few of the notable combinations
of central/distributed components deserve mention:

**Centralized** architecture, where the authoritative, complete copy of the data it maintained at the provider, and clients cannot keep their own;

**Storage + Rendezvous** enables clients to find one another, while also providing the authoritative copy of the most recent data available, functioning as a *primary site*;

**Rendezvous** only, providing a simple mechanism for clients to find one another, but no storage;

**Decentralized/Peer-to-Peer (P2P)** architecture, where clients employ manual rendezvous through direct Internet addressing or equivalent to find one another and communicate.

These examples are not exhaustive, but do provide considerable breadth in terms of the kinds of applications we support. Even in the completely centralized case, where the nature of the application may not allow for clients to maintain the authoritative copy of their data, it is possible to provide data and metadata security with the cooperation of the provider. The less unencrypted data a provider can see, the more difficult indexing becomes—and with it, related services such as searching and advertising, that depend on a central view of the data. There may be, for example, some degraded search result quality, although client-side indexing is a feasible approach. Inspired by the models set forth in Privad [40] and RePriv [39], we believe advertising could be offered to users without a loss of their privacy. Developer challenges stemming from the adoption of our model are not unsurmountable, and only apply to a subset of functionality applications may be interested in providing.

For the purposes of this work, we assume the availability of a storage and rendezvous node, a model that echoes many of today’s cloud storage providers.
3.3 Design Goals

Subsequent to our examination of the types of supported applications and their respective infrastructure, we extract a set of goals for our system.

Data and Metadata Security
The provider should not be able to intercept the encryption keys for a user, or infer anything about that user’s data. In some fully-centralized instances this is only possible with the application’s cooperation, and would involve mechanisms outside the scope of our work. In many instances, however, this simply means that all encryption-related operations need to happen at the client.

Simple Key Management
Applied cryptography’s perhaps greatest usability failure remains the difficulty of achieving robust key management. For CRYPSTOR, key mismanagement would render the security promises made baseless and give end-users a false sense of confidence, or lead to data loss due to the inability to decrypt when keys are lost, an egregious violation of the integrity guarantees otherwise expected from the storage layer.

Sharing Capabilities
The need to share data between users goes back to early operating systems designs, and is so intertwined with everything computers are used for today that it would simply not be practical to have a system unable to support it.

Efficiency
Our system must efficiently make use of disk space, and minimize the size and number of network requests as aggressively as possible, avoiding gratuitous work. Encryption is a cost in terms of resource usage, but since the workload is distributed among all clients, we expect a reduced overall strain on the system.
Chapter 4

Cryptographic Engineering

CRYPSOR merges a number of existing cryptographic building blocks in order to achieve its intended security properties. We describe these blocks here, and how they are combined and used.

4.1 Building Blocks

Our system requires algorithms for hashing, symmetric encryption, symmetric message signing, public key cryptography, and key stretching. We also assume the availability of a cryptographically-secure Secure Pseudo-Random Number Generator (SPRNG) which is a minimum requirement for making any cryptographic operation possible.

**Hashing** is done via the Secure Hash Function, 384-bit digest (SHA-384) [33].

**Symmetric authenticated encryption** is carried out using the Advanced Encryption Standard, 256-bit key (AES-256) in Galois/Counter Mode (GCM) [36]. **Rijndael** is the current algorithm of the standard [1,34].

**Symmetric signing** is accomplished by Hash-based Message Authentication Codes (HMACs) [43].

**Our Public Key Cryptosystem (PKCS)** is RSAES-OAEP[41]: Rivest, Shamir, & Adleman (RSA) encryption used with Optimal Asymmetric Encryption Padding (OAEP).
Key stretching is realized by applying scrypt [4] to the user’s password; scrypt is chosen as our key derivation function (KDF) because it imposes a memory-bound cost on the derivation process, which provides better resistance against brute force attacks than a strictly computationally-intensive function such as the Password-Based Key Derivation Function 2 (PBKDF2) would. The resulting material is referred to as a derived key.

All of the selected building blocks are used in a manner that allows for easy replacement of any one of them, if attacks are discovered in the future. While the choice of algorithms is not sufficient to ensure either confidentiality or security, a poor selection for these basic elements at the design stage is going to lead to critical weaknesses that ultimately render any cryptographic operations useless, mere nuisances to potential attackers. Considered another way, it is absolutely necessary, though not sufficient, that our system uses the best current cryptographic components.

4.2 Constructs

To keep things simple for the user, we base all of the cryptographic keys used throughout the system on one password, and use key stretching [42] to derive keys with increased brute force resistance (leveraging the computational requirements imposed by the KDF). The system workflow when a new user key is generated is shown in Figure 4.1.

The user’s password is fed as input to the scrypt function, in conjunction with a randomly-generated salt. The derived key is used as a wrapper key \( W_k \) around the master encryption key \( M_k \). Using keys in this manner allows the user to change their password without having to re-encrypt all of the data. \( M_k \) is generated by the SPRNG and can be changed at any time, though such replacement would require full re-encryption of the user’s files.

A public/private key pair, to be used with RSA Encryption Scheme (RSAES-OAEP), is also generated from the SPRNG. The public key \( P_k \) is saved to the backend storage, with the intention of making it available to
other users of the system. The private (secret) key $S_k$ and the base encryption key $M_k$ are encrypted via AES-256 under the wrapper key $W_k$, and their cyphertexts are also saved to the backend storage.

$$E_M = encrypt(M_k, W_k)$$

$$E_S = encrypt(S_k, W_k)$$

We proceed in a similar fashion whenever a group encryption key is required for data to be shared between different users, as shown in Figure 4.2. The initiating party randomly creates a symmetric group key $G_k$ and encrypts it with the public key of each of the intended recipients, to obtain

$$E_{G_i} = encrypt(G_k, P_k(u_i))$$

1We do not discuss Public Key Infrastructure (PKI) in this work, and do not claim to provide a mechanism that is sufficient for the trusted binding needed for such infrastructure.
where $U_i$ is another user in the system and $P_k(U_i)$ represents that user’s public key. The encrypted material is then shared with the other participating users via a secondary channel (e.g., by publishing on a website or sending via email). The recipient can easily decrypt $E_{G_k}$ with their own $S_k$ and access the shared data. Other participants can be similarly invited by anyone with read access to the shared data, by further disseminating the group key.

When access must be revoked for a subset of users (e.g., at time $T_{revoke}$), a new group key $G'_k$ can be generated and distributed to the new set of users as per the above, and used from $T_{revoke}$ onward. The old $G_k$ is encrypted under $G'_k$ and stored as metadata, giving current and future users seamless access to all data that has not been re-encrypted yet, possibly because only read operations have been performed. Users that have been previously part of the group can still read all of the data they were previously granted access to, as they still hold $G_k$. To also revoke read access, all of the information must be re-encrypted, though the user may have made local copies of the data before $T_{revoke}$ anyway.
Chapter 5

System

We implemented an example application in order to demonstrate the feasibility of our system, and to better understand any trade-offs or limitations the model may have. Our system is split into a server that ensures ordering and availability, and a client that runs on user’s devices and performs the necessary operations on the file data and metadata.

5.1 Server

A direct consequence of our system’s intent to not leak any useful private information to the server is that both metadata and file contents must be kept confidential as they pass through on their way to end-storage or to other client instances. The server itself, then, does little more than accept encrypted blobs, order them, and pass them back to the clients that request them. The simplicity of the server ensures that it can be run in many environments, and in fact it could simply be a gateway into the back-end storage, but need not be more tightly integrated than that. For example, a user may even be able to use S3 for storage, and run the server component on a different provider’s platform, or on a computer they own and know to have a reliable connection. This mode of operation is likely impractical due to the lack of low-latency and high-reliability guarantees inherent in home device deployments. We speculate that the rise of low-cost, low-
power devices like the RaspberryPi [22] will extend the market for simple and cheap or free [13] collocation facilities for this class of computers. In that scenario, further decoupling the server from the storage is highly desirable, as it would give the user even more control over where and how their code is being run.

The server is implemented using the Twisted [27] Python framework in a straightforward manner. The ordering is maintained by naming the blobs with an identifier that is derived from a monotonically-increasing counter and the order of their receipt by the server. If the server is restarted, or a migration happens, the state needed to resume operation resides entirely on the end-storage. Synchronization happens whenever clients join with the server and send a message containing the last received blob sequence identifier, followed by a list of their pending updates. While clients are ‘online’, the server informs them immediately of any changes it has processed.

5.2 Client

Our example application is modelled after existing personal cloud storage software, such as Dropbox [12] and SpiderOak [25]. At a high-level, the application allows users to specify one or more directories on disk that should be synchronized across client instances, and then monitors these directories for file changes. When a change is detected, the client encrypts the file and uploads the cyphertext to the server. Incoming changes are processed by decrypting the file to a temporary location, and (re)placing it inside the appropriate monitored directory.

The client runs in the background on the user’s machine, contains three subsystems as shown in Figure 5.1 and some glue code that provides general functionality and interacts with the user’s operating system on their behalf.

5.2.1 Monitoring

Directory monitoring is performed using the native notification mechanism of the user’s file system through the watchdog [29] third- party library.
The client registers interest for a particular path (that it received from the user) and waits in the background to be notified of file system events. The notification that arrives contains the type of event (addition, movement) and the full path to the file system entity that was acted upon. The monitor component communicates this event as a tuple of OP:path (where OP indicates the type of operation performed on the file) across a ØMQ [30] inter-process communication (IPC) pipe, to the crypto module.

### 5.2.2 Cryptography

The cryptographic module is implemented in the C programming language, in order to obtain high performance for our encryption and decryption operations. It functions as a small daemon and communicates with the rest
of the client over IPC, as previously described. On local file changes, this module encrypts the files and their metadata with the appropriate key, and informing the transport module of the location of the blob to be sent to the server. On remote (incoming) changes, it informs the client glue of the path to the location of the received blobs, after they have been decrypted. The glue then uses the metadata to decide which file to act upon—in terms of the ‘real’ path inside the monitored directory—and what operation to perform. Currently, only deletions, stat changes, and replacements (integral uploads/downloads of file contents) are supported.

5.2.3 Transport

The transport module is responsible for communication with the remote endpoint, implementing the Twisted API that our server expects. It receives paths from the client glue or from the upstream, and forwards them along as appropriate. For disconnected operation, it also maintains a list of all the local changes that are pending transmission—we opt to store all of them, rather than the most recent version of any given file, in order to provide clients with more version information and backups of their file sets.
Chapter 6

Evaluation

Evaluation of the proposed design as implemented in the prototypical system from Chapter 5 requires understanding of its security, performance, and usability.

6.1 Security

A traditional view taken in conversations about security is that an inherent trade-off exists between usability and absolute security. On the one hand, defences should be layered in order to minimize the likelihood of compromise, while on the other the more difficult it becomes for users to perform the tasks they are interested in, the more likely they are to try and short-circuit or altogether circumvent the protections in place. For our system, extraneous difficulties would ultimately prevent developers and users from adopting this model. This reasoning motivated us to build all of the cryptographic functionality from a singular user password, although it is acknowledged that passwords have numerous issues in their own right. In our threat model, the storage provider is honest but curious, meaning that while it does not try to maliciously interfere with its clients, it is interested in data it stores and will attempt to glean as much information from it as it can. Given our current design, we can assume that the provider will keep copies of most if not all user data, particularly the wrapped $M_k$. 
In our current model, we expose:

- access patterns that may leak information regarding the popularity of certain objects;
- the participants in a group share;
- the user’s symmetrically-encrypted base key

Of these, the last item deserves more discussion. In extant models of cloud applications, only one password allows access to the user’s data. Even if an interested party were to obtain all of the cryptographic hashes of the user’s password in the hopes of breaking one of them at a later date, or otherwise obtaining a reused password from a leaked source, it would not be able to access the underlying information if it was not the current password they obtained. In our model, since the effective key $M_k$ is encrypted under the password-derived wrapper key $W_k$, compromising any of the user’s passwords may give access to $M_k$ and the ability to decrypt all of the data. The best approach for safeguarding the key would be to store it on a smart card, though this unfortunately comes close to being very unusable for most users. In keeping with our password-only, online model, it is possible to stripe the key(s) across multiple, non-colluding providers, in a way that provides both redundancy against data loss, and better resilience against the attack as described. This still carries a usability overhead for the user, as multiple storage providers need to be setup and configured, but it may be a simple enough process that it would not cause significant user drop-out.

6.2 Performance

We analyze the performance overhead of our system by measuring the time required for the file encryption and decryption operations. The machine used for testing is an Intel Core 2 Quad Q9400 @ 2.66 GHz with 8 GB RAM and a 7200 RPM HDD. We run 100 trials each of the encryption and decryption operations on files of different sizes, representative of a selection
Figure 6.1: Overheads of the cryptographic operations.

of source code, photos, music, a Linux distribution disc image, and a video. Figure 6.1 shows the plot of these timings. The time required for encryption and decryption scales mostly linearly with the size of the input, which is something we intuitively expect as every file must be visited in its entirety.

In terms of bandwidth overheads, we only incur the penalty of encapsulating metadata inside our blobs, including file deletions which have to be signalled to other clients in this manner.

6.3 Usability

We can claim with confidence that our design does not increase complexity for users beyond what is typical of existing comparable services. Only a username and a password are needed, a requirement which is consistent with what current systems ask of users for authentication and authorization. Alternate designs, such as those suggested in Chapter 3 that lack a central node, would carry additional complexity, as the burden of configuration must be pushed to end-users.
Chapter 7

Future Work

7.1 Security Hardening

Striping $M_k$ across a few providers, and keeping only a small encrypted file with pointers to these other data repositories is the first step toward mitigation of the weakness described in Section 6.1. With every password change, the file itself is to be re-encrypted, and the order and structure of the stripes of $M_k$ would be unknown to a potential attacker. Older versions of this file would contain only out-of-date information with respect to key striping, at most disclosing the providers where the stripes are held, but not how they should be used for reconstruction of $M_k$.

The system should also easily provide mechanisms for using $M_k$ from a user-supplied device, thus obviating the need to store the encrypted version of this key in any provider's storage.

7.2 Performance

While we do not incur a significant bandwidth overhead compared to a version of our system without encryption, further reductions are possible by only uploading deltas of files instead of the entire file contents. Storing file modification in the spirit of operational transformations [37], instead of entire snapshots, allows for better use of the provider storage space, which
we assume the user is paying for and thus has a motivation to use effectively. Periodically, snapshots would be taken so that files can be recreated on new devices quicker than if the entire transformation log were replayed.

Another way to improve performance is to reduce the number of data copies made in memory or on disk. In our current implementation, we encrypt first and then transfer the data over the network; it would reduce time and client-side storage cost if the encrypted data was streamed directly from the client application instead of saving it to the HDD first. Moving the transport and cryptographic modules closer to one another would help with performance in this regard.

7.3 Replacing Cryptographic Components

The cryptographic constructs selected for CRYPSTOR are not known to be vulnerable to any attacks. Nonetheless, it is important to be able to replace any of the chosen cryptographic functions if there is a compelling reason to do so, and our design allows this at the cost of re-encrypting the affected data. For example, if AES-256 is shown to have vulnerabilities, it can be replaced with Serpent [32], one of the other finalists in the original National Institute of Standards and Technology (NIST) request for proposals [47]. The components fit in equivalence classes as outlined in Section 4.1. Additional engineering may be required to match APIs between the new and old components, in order to make them fit, and most importantly diligence is required to ensure the combination of cryptographic components does not lead to undesirable or unforeseen consequences with regards to the overall cryptographic security of the system.

7.4 Privacy

The central application server currently knows all of the usernames in the system. While we have not commented on the need for a PKI, the applica-

---

1Where encryption is concerned, an attack is understood to be a mechanism that allows total or partial information leakage in shorter time than it would be required to bruteforce the encryption key(s).
tion server is effectively serving this role by storing all of the known public keys and making them available for group sharing, while also enforcing uniqueness among users. This requirement can be removed if another PKI is available, or if users employ a secondary channel for key dissemination. In that scenario, storage is agnostic to the clients using it, and will serve any issued requests; clients that have the group share key can make sense of the contents, while unauthorized viewers cannot.

7.5 Group Shares

Revocation of group encryption keys does not automatically prevent users from accessing the data they previously had read permissions for. Since it is always possible for users to make copies of the data they can ‘see’, no attempts are made to solve this problem definitively in the system. The only guarantees made are regarding new data entering the system, namely the fact that users that do not hold the new group encryption key would not be able to access it, irrespective of whether they could read previous versions of it.

In terms of preventing dissemination of the group encryption key to parties outside the initiator’s list, a challenge mechanism can be employed that would require some operations to be signed with a user’s $S_k$, which we assume they have no interest in sharing with others. To prevent simple man-in-the-middle attacks, the sequence of requests that need to be signed can itself be sent out, encrypted with the $P_k$ of the user. Whenever a challenge is failed, the server’s policy can decide what to do, e.g., the server can re-key the share, and exclude the user that owns $S_k$ from the new share. More advanced approaches can be considered for this enforcement and prevention protocol.

7.6 Removing Centralization Requirements

Centralized services benefit from having a simpler bootstrapping process than fully-decentralized ones can achieve, as well as simpler conflict resolution. Having a known rendezvous point for all clients to use makes setup
and node discovery trivial. Ordering of operations—the system’s *ground truth*—can also be reliably maintained by this primary site, even if the data and metadata are confidential. However, centralization is a bottleneck for performance and availability, as the overall system health depends on the primary site. There is considerable literature on decentralization and P2P architectures that can be leveraged for a more robust design, one that either uses a central node in its default mode but can fallback to a distributed system if needed, or altogether does away with an explicit primary site.
Chapter 8

Conclusion

It is important to acknowledge that economic, rather than purely technical, reasons are central to the issues we identified in regards to the current cloud storage landscape. Storage providers aim to minimize client resource usage as a means of maximizing their own profits, which often translates to deduplication. The ability to mine user data and resell aggregate information can be another revenue stream. The use of encryption complicates both of those matters and possibly cuts off the additional revenue streams, but serves the users’ interests and as such we consider it to be a more desirable property to guarantee and enforce in the system.

We have designed a system that provides confidentiality and security for user data, built a prototype of our design, and evaluated its performance with respect to the cryptographic operations it performs most frequently. A natural extension of prior work, our system allows for disconnected operation and handles the storage and synchronisation of generic document hierarchies, such as files and folders. CRYPTOR reinforces our original hypothesis that data security is achievable in the cloud computing environment. While our system still requires further optimizations before it would be suitable for a general deployment, it demonstrates that high levels of confidentiality and security for data stored online can be reasonably achieved.
Bibliography


https://blog.dropbox.com/2011/06/yesterdays-authentication-bug/, →
pages 2, 9

[3] Four Reasons We Choose Amazons Cloud as Our Computing
netflix.com/2010/12/four-reasons-we-choose-amazons-cloud-as.html →
pages 1

[4] The scrypt Password-Based Key Derivation Function (draft 01),
→ pages 14

https://aws.amazon.com/s3/, → pages 2, 3

[6] Client-Side Data Encryption with the AWS SDK for Java and Amazon
latest/dev/UsingClientSideEncryption.html → pages 4

http://aws.amazon.com/sns/, → pages 2


[9] ClearDB — The Ultra Reliable, Globally Distributed Cloud Database
For Your MySQL Applications, November 2012.
https://www.cleardb.com/ → pages 2
http://www.cloudamqp.com/ → pages 2

http://www.credant.com/cloudsecurity/ → pages 6

https://www.dropbox.com/ → pages 3, 18


http://www.eucalyptus.com/ → pages 9


*ACM SIGMOD Record*, 1989. → pages 5, 24


32
