SteelDoor: Secure cloud storage

Author: Maarten Abbink
m.abbink@student.tudelft.nl
Supervisors: Bogdan Ghit, Alexandru Iosup, Dick Epema
{B.I.Ghit,A.Iosup,D.H.J.Epema}@tudelft.nl

Abstract

In this report we propose a system for securely storing files online, named SteelDoor. SteelDoor is a software service that safely stores end users' encrypted files "in the cloud", on storage facilities provided by Amazon Web Services (AWS). Encrypted files submitted to the SteelDoor system are impossible to decrypt by the system operator or the cloud infrastructure operator. Based on state of the art asymmetrical encryption technology, only the end user will be able to ever decrypt a file retrieved back from the SteelDoor system.
With encryption and decryption of data happening solely on the end user's machine, SteelDoor consists of two parts, an online software service responsible for replicated storage, and a client application responsible for all cryptography, submission and retrieval of files and meta data. The client application communicates asynchronously with the service through HTTP requests over a secure (HTTPS) connection.
Please note that due to time constraints only a small portion of the system is implemented. However, a complete model of the required deliverables is described in this document, including unimplemented aspects.

Introduction

In recent years the term "cloud storage" has increased in popularity to the point that regular computer users with no particular interest in the science behind it have taken an interest in it. It is this popularity that has made end users aware of the benefits of cloud storage over traditional data storage solutions, which in turn has led to the advent of a variety of cloud storage providers.
Current cloud storage providers include Dropbox[1], Google Drive[2] and Microsoft SkyDrive[3]. Although these promise to keep their users' files safe from unauthorized third parties, they explicitly do read these files themselves to provide end user utility. Alternatively, users can pick from a variety of security-centered cloud storage providers, like SpiderOak[4], CryptoHeaven[5] or mozy[6]. There also exist solutions that convert insecure cloud storage services into secure ones. Applications like BoxCryptor[7], CloudFogger[8] and EncFS[9] take care of encrypting all files on a user's computer, before any data is sent to the cloud storage provider. Decrypting said files happens again after retrieval from the cloud, leaving the cloud storage provider oblivious to any actual information inside the files it is holding for such a

[1] http://dropbox.com
[2] http://drive.google.com
[3] http://skydrive.live.com
[4] http://spideroak.com
[5] http://cryptoheaven.com
[6] http://mozy.com
[7] http://boxcryptor.com
[8] http://cloudfogger.com
[9] http://www.arg0.net/encfs

user. Even in the scientific community secure cloud storage has received some attention. In [1] the authors propose a design for implementing secure cloud storage on an untrusted infrastructure. Besides security, the data structures involved in implementing cloud storage systems require have received attention as well [2,3].
In this work we will propose and implement the SteelDoor secure cloud storage system on the AWS cloud computing infrastructure. Besides serving as a practical exercise in the area of cloud computing, the goals of the system are to provide a dependable, highly available generic cloud storage service that can function without any knowledge of user-owned data. Co-designed with the storage service is a client application in charge of encrypting and decrypting files and meta data. The client application enables end users to manage the files and associated meta data stored in the cloud in a comprehensible way, all the while being able to quietly encrypt all information sent to the storage service.
The remainder of this report is structured as follows: In Background a brief summary of the SteelDoor requirements is presented. Then the system design is presented, focusing on the data structures of the system. The subsequent section Experimental Results addresses approaches to verify the appropriate fulfillment of mentioned requirements and how cloud infrastructure interactions support this goal, which is followed by a discussion of the SteelDoor system and our conclusion.

Background

The SteelDoor service must be capable of holding a seemingly infinite volume of user-supplied files. These files can be of any size, ranging from only a few bytes to multiple gigabytes. At the same time servers must always respond to clients' requests in a timely manner, despite the fact that one single server can only achieve a maximum data throughput at any given moment. To allow one user to connect with multiple clients, a mechanism capable of efficiently communicating changes to the user's file collection is required, keeping in mind that the system should function properly without any knowledge of a user's files - or even file meta data.
Even when deploying on a cloud infrastructure, replication can be beneficial to the system's performance. Not only can replication avoid inaccessibility, but it can also help spread resource accessing load. For these reasons we want persistent storage to be replicated multiple times, and the servers responding to clients' requests to work in unison, all transparent to the outside world.
Finally, throughout the system's lifetime, the system should be able to maintain itself, scaling up and down as required by changing user demand. In order to minimize cost, we want to always operate using only slightly more than the minimum amount of resources needed for smooth operation.

System Design

Resource Management

Files stored in the SteelDoor system are assigned a globally unique 64 bit identifier. Each server keeps all names used for the files it currently maintains in memory. In the remainder of their memory, the servers maintain a distributed hash table (DHT) that stores all globally used names and their current storage locations. This way, if a server is asked to serve a file it does not have itself, it queries the DHT for the current file's storage locations and responds with a redirect to a randomly chosen correct location. To allow storing seemingly endless files, data is written into so-called container files. This eliminates unnecessary IO for file

metadata and scales better with the use of fixed-size leased block storage. Each container file is located on a fixed-size leased storage volume and can grow to the size of that volume. Files that don't fit in (the unused remainder of) one container are broken up into pages that then become retrievable separately. Once a container fills up, it is marked as sealed, meaning that no more data can be appended to it.
Each server has at most two unsealed containers files: One that is actively being written to, and another to quickly switch to when the first one fills up. All other container files a server has are marked as sealed (i.e. full). Data can not be modified, only deleted and rewritten; files that are deleted are marked as such. Container files holding deleted files are periodically consolidated to free up virtually unused space.
For each container file that a server manages, it maintains a throughput measurement. When the combined throughput for a server increases beyond it's limit, one or more containers are relocated to less busy servers. If no sufficiently idle server is available a new instance is leased from the cloud infrastructure.

System Policies

The components providing the SteelDoor service (servers and storage volumes) communicate in a way similar to what is known as the actor model, both internally and with the outside world. This means that incoming request will be held in a priority queue and then answered sequentially. The length of such a resource's queue can then be used to determine whether or not the resource is able to satisfy all requests in time. Additionally, system-internal communication can easily be prioritized.
The publicly exposed SteelDoor service is accessible through every server involved in it. Requests for reading or writing can be answered by any SteelDoor server, but oftentimes a server will not be able to satisfy such a request directly, as it does not manage the resources required itself. Instead, by querying the namespace DHT, a server will always be able to respond with a redirection to the the server that will be able to directly answer the request. This model facilitates load balancing without the need for a dedicated load balancing service. Simultaneously, it also does not exclude such an addition to the service stack when is appears to be necessary.

Additional System Features

To provide a system for clients to retrieve their stored files on a client that does not know any of file identifiers used for those files, SteelDoor provides three file concepts: regular files, growing files and key files, all of which are distinguished by a globally unique identifier and secured with a semi-secret cookie. Every data submission can automaticaly be broken up into multiple segments: the head and subsequent tail segments, each of which will have their own a unique identifier. When regular files are submitted to SteelDoor, the client receives back the head- and tail-identifiers used for it, as well as the shared cookie for the file. The cookie is in fact an identifier for a key file. A key file stores an asymmetrically encrypted symmetric encryption key for a regular file. The key file's cookie is determined by the client. Growing files have a leading section (possibly broken up into head and tail segments), followed by zero or more ordered changeset sections (also possibly broken up). The difference between the head/tail decomposition of data segments and the lead/changeset composition of growing files is that coherent head/tail sections use the same cookie. The sequencing is managed by the server, which then can prevent a growing file from getting too many changesets. The leading and changeset sections of a growing file each receive their own cookie, which makes them independently readable if encrypted. Every user maintains a growing file containing all their files' identifiers, private meta data, cookies and keys. This

way, the server only needs to maintain a user's file index stored as a single growing file, from which a client can retrieve all information necessary to access all user submitted data in the system.
Apart from a user's file index, the service also keeps track of user's registered clients applications. Each client is represented by an access token and a public key. For each request a client makes, it sends along its plain access token, and the required file identifier and cookie, both asymmetrically encrypted with it's private key. Only if the decrypted file identifier and cookie match, access to the requested data is granted. Multiple clients are supported by having an already authenticated client add encrypted versions of keys for a different client.

Experimental Results

Experimental Setup

The SteelDoor system is intended to run on the Amazon Web Services (AWS) cloud computing infrastructure. This choice is arbitrary, although strongly guided by the perceived lack of useful documentation for the DAS system.
Since the scope for this assignment is wildly disproportionate to the amount of work supposed to go into it, we did not have enough time to successfully implement any automated interaction with AWS. For this reason this section will not discuss computational experiments, but in its place a theoretical discussion of the capabilities and limitations of the SteelDoor system will be addressed.

SteelDoor properties

The namespace of SteelDoor files spans 64 bits. For every file, a cookie/key file is reserved, leaving 63 bits for addressing actual files. Assuming that no more than 10% of files are split into head/tail sections, all of which do not use more than one tail section, this leaves over $8.3*10^{18}$ unique identifiers for actual files. If 14 billion ($14*10^9$) users (two accounts for each human on earth) were to use the SteelDoor system, an even distribution of available identifiers would accredit each user over 8.3 billion ($8.3*10^9$) identifiers.
Assuming a file container size of ten GiB ($10*2^{30}$ bytes) and a maximum overhead of 100 bytes per identifier, the 63 bit data namespace caters for a maximum storage capacity of 81920 YiB ($81920*2^{80}$ bytes), requiring an additional 800 EiB ($800*2^{60}$ bytes) for overhead.
More realistically, assuming an average file size of 5 MiB ($5*2^{20}$ bytes), the total storage required would be 40 YiB ($5*2^{83}$), not counting the 800 EiB of overhead required for this number of files. Including overhead, one container file can hold 2047 such files. Assuming that each file takes up ten eight-byte words (80 bytes) in the owning server's memory, a server equipped with 16 GiB ($16*2^{30}$ bytes) of memory which is used entirely for tracking managed files, can track a little over $2.1*10^8$ files, limiting the number of ten GiB containers it could sensibly manage to 104,908. If the same amount of memory per file is required to represent it in the DHT, and a three-fold replication of container files is used, a sensible minimum of three servers equipped with 16 GiB of memory each, can collectively maintain 161 million files. Including overhead, this is the equivalent of 736 TiB in 5 MiB files on 78,127 10 GiB containers.
With current AWS Elastic Block Storage (EBS) pricing, the cost for leased storage volumes increases in steps of $1.10 per month, for each ten GiB volume. Current AWS Elastic Computing (EC2) machines cost between $61.20 and $374.12 per month, depending on their specifications. Data transfer caused by retrieving data from the SteelDoor system depends costs $1200 per month for the first 10 TB. Beyond that costs climb less steeply as

larger intervals are reached.

Discussion

A new large scale system for which the success cannot be predicted with great certainty, large upfront commitments, such as hardware expenses, are a great entrepreneurial risk. This is especially true for a system that is capable of storing yobibytes of data, where a provisional hardware purchase can easily be far too large for the eventual demand.
By using the AWS cloud infrastructure, this entrepreneurial risk of a startup is simply eliminated, making it an attractive option compared to in-house resource acquisition. The variety of services that make up AWS make it rather simple to quickly develop complex, yet reliable systems. Since these services were developed to support many different developer users, even the commonly large upfront commitment of developing complex infrastructure-supporting tools is eliminated. On the other hand, as a cloud-based system grows and gains traction, a versatile cloud infrastructure like AWS offers the chance to replace the use of costly hosted services, such as load balancing and monitoring, by proprietary solutions.

In terms of cost, operating a newly conceived system like SteelDoor on the AWS infrastructure will likely be within the bounds of $2000 per month, as long as the user base stays within a few thousand. As soon as it grows beyond that, we suspect that it will make sense to move towards an increasingly in-house solution. First by leasing physical machines in existing data centers and privately administering them. The next step could be leasing only rack space and utilities from an existing data center, but furnishing it with private hardware. The final step towards a truly proprietary infrastructure would be to build a private data center.
Since we can neither predict a rate of growth for the academically purposed SteelDoor system, nor can we make accurate predictions about the cost of of leasing and purchasing hardware, we cannot indicate a user base that would be required to justify any of these steps.

Conclusion

When working on this assignment, and consequently on the SteelDoor system, numerous setbacks limited the visible output of the effort put into it. These problems include collaboration deficiencies with lab partners, little or impractical documentation for the cloud infrastructures at our disposal and the very limited time to design a complex system as well as implement it in order to conduct computational experiments.
The explicit ban on using existing technology for integral parts of the to-be-designed system proved to be both a challenge and a limitation to what is achievable. The challenge this ban presented was that the parts of the system that were to be implemented had to provide educational value with respect to the cloud computing paradigm. This is why the (incomplete) SteelDoor implementation currently only covers the fundamental data structures required to build a large scale cloud storage system. Since this decision also lead to not strongly focusing on getting the AWS api under control, eventually no (autonomously) working system could be implemented in time.

References

[1] S. Kamara and K. Lauter. "Cryptographic cloud storage." *Financial Cryptography and Data Security* (2010): 136-149.

[2] B. Calder et al. "Windows Azure Storage: a highly available cloud storage service with strong consistency." *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.* ACM, 2011.
[3] Beaver, Doug, et al. "Finding a needle in Haystack: Facebook's photo storage." *Proc. 9th USENIX OSDI* (2010).

Appendix A: Time Sheets

Activity	Date	"think-time"	"dev-time"	"xp-time"	"analysis-time"	"write-time"	"wasted-time"	total-time	total per day	comment
Getting started with AWS	2012-11-19						9.25	9.25	9.25	problems with credit card validation, studying AWS docs
Getting started with AWS	2012-11-20						1.75	1.75		reading AWS docs
Determining scope of system	2012-11-20	5					2.5	7.5	9.25	asking security forum about encryption techniques
Determining scope of system	2012-11-21	5					3	8	8	refining idea to fit assignment objectives, researching cryptography
Determining scope of system	2012-11-22	4					4.5	8.5	8.5	refining idea, studying AWS docs
writing report	2012-11-23					8.75		8.75	8.75	structuring report, looking into related work
Designing scalable storage system	2012-11-26	7					1.25	8.25	8.25	
Designing scalable storage system	2012-11-27	8.5						8.5	8.5	
Designing scalable storage system	2012-11-28	6.75						6.75	6.75	
Designing scalable storage system	2012-11-29	7.25						7.25	7.25	
Designing scalable storage system	2012-11-30	4.5						4.5	4.5	
Designing scalable storage system	2012-12-01	2.5						2.5	2.5	
Designing scalable storage system	2012-12-02	3						3	3	
set up code repo and web stack	2012-12-03	1.5	7.25					8.75	8.75	
implementing data structures	2012-12-04		8.5					8.5	8.5	
implementing data structures	2012-12-05		7.75				1	8.75	8.75	
implementing data structures	2012-12-06		8					8	8	
implementing data structures	2012-12-07		7.25				0.75	8	8	
implementing data structures	2012-12-08		2					2	2	
implementing data structures	2012-12-09		6					6	6	
implementing data structures	2012-12-10		9.5					9.5	9.5	
implementing data structures	2012-12-11		9.5					9.5	9.5	
implementing data structures	2012-12-12		9.25					9.25	9.25	
implementing data structures	2012-12-13		9.5					9.5	9.5	
writing report	2012-12-14					9.5		9.5	9.5	
	2014-12-15									
	Totals:	55	84.5	0	0	18.25	24	**181.75**		
							checksum:	181.75		