ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

L4 Security of Computer Systems

Project Report

"Evaluating data encryption on locally hosted open source Amazon S3 compatible object storage platform"

Evangelos Dimoulis, AM: 421

Ioannina, Greece February 2021

1. Introduction

Object storage, also known as object-based storage, is a strategy that manages and manipulates data storage as distinct units, called objects. These objects are kept in a single storehouse and are not ingrained in files inside other folders. Instead, object storage combines the pieces of data that make up a file, adds all its relevant metadata to that file, and attaches a custom identifier.

MinIO [1] is a popular open-source object storage server compatible with the Amazon S3 cloud storage [2]. Applications that have been configured to talk to Amazon S3 can also be configured to talk to MinIO, allowing MinIO to be a viable alternative to S3 to gain more control over the object storage server. The service stores unstructured data such as photos, videos, log files, backups, and container/VM images, and can even provide a single object storage server that pools multiple drives spread across many servers. To access the MinIO server though a client via a HTTP request, MinIO implements Server-Side Encryption (SSE) requiring TLS/HTTPS communication between the client and the server in order to ensure data confidentiality and integrity. However, another important aspect of securing the data that resides in MinIO server is in what form the data is ultimately stored on the object store and if some sort of encryption is applied to the stored objects and the object metadata.

Our evaluation will focus on studying the security guarantees that a locally hosted MinIO instance offers, upload files via a client to the object store, and if the data is stored in raw, we will introduce an encryption scheme to secure the uploaded data against storage provider compromise, via traditional encryption algorithms (e.g., AES, AES-CTR). We will report the performance of the encryption applied on the uploaded files, and if possible re-encrypt the uploaded files online in case we want to revoke access to a user or service that could previously access the data.

2. Background

In this section we briefly describe the functionality of object storage systems and the benefits gained by leveraging this kind of storage. Furthermore, we refer to Amazon S3 Object Storage and how objects are stored in Amazon S3.

2.1 Object Storage System

Object based storage system [3] is a data storage system which process data as objects when compared with other state of the art distributed storage system as blocks and files. Both files and block storage is converged to called as Objects [4]. An object is variable-length not a fixed size like blocks, and can be used to store all type of data, such as files, database records, audio/video images, medical record. A single object could even be used to store an entire file system or database. The storage application decides what gets stored in an object. Unlike block I/O, creating objects on a storage device is accomplished through a rich interface similar to a file system. Object storage stores digital data, machine data, sensor data and easily manage files, metadata, and having very high scalability. Keeping track of content objects using these metadata descriptors and policies makes access, discovery, replication, distribution, and retention much more practical than in traditional approaches.

2.2 Benefits of Object Storage

There are many reasons to consider an object-storage-based solution to store massive volumes of data. Object storage is used for storing/managing large volumes unstructured data such as images, video files or even archived backups which are data types that are typically static but may be required at any time. Scalability and reduced complexity are two of the main advantages of object storage. Objects or discrete units of data are stored in a structurally flat data environment within the storage cluster and by adding more servers to an storage cluster, higher throughput and additional processing capabilities can be supported given that object storage also removes the complexity that comes with a hierarchical file system with folders and directories. Furthermore, object storage goes hand in hand with cloud or hosted environments that deliver multi-tenant storage as a service allowing many companies or departments within company to share same storage repository. Another aspect is that, object storage services follow a pay-as-you go pricing strategies, and the pricing is usually volume-based, which means that a user or company pays less for very large volumes of data rendering this kind of storage more affordable [5].

2.3 Amazon S3

Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. Customers of all sizes and industries can use it to store

and protect any amount of data for a range of use cases, such as data lakes, websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics. Amazon S3 provides easy-to-use management features in order to organize data and configure finely-tuned access controls to meet specific business, organizational, and compliance requirements. In Amazon S3, objects are uploaded and stored into buckets. Amazon S3 provides REST APIs for creating and managing buckets which requires writing code and authenticating the requests. The requests are initiated and executed via clients [6]. Alternatively, managing buckets and objects can be achieved using high-level S3 commands provided by command line tools used to establish communication to an Amazon S3 instance.

2.4 Amazon S3 Objects

Amazon S3 is essentially an object store that uses unique key-values to store as may object as a client desires. Objects can be handled in one or more buckets and each object can be up to 5 TB in size.

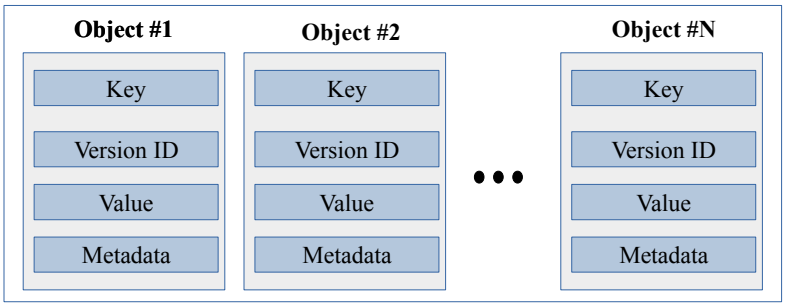

Figure 1: A collection of objects stored in a bucket

Figure 1 illustrates an example buckets which consists of a collection of total N objects. Each object is composed of the following (basic) attributes which uniquely describe the object in a bucket:

- **Key:** The name that you assign to an object. You use the object key to retrieve the object.
- **Version ID:** Within a bucket, a key and version ID uniquely identify an object.
- **Value:** The content that you are storing. An object value can be any sequence of bytes. Objects can range in size from zero to 5 TB.
- **Metadata:** A set of name-value pairs with which you can store information regarding the object.

3. MinIO

MinIO is a High Performance Object Storage released under Apache License v2.0. It is API compatible with Amazon S3 cloud storage service. MinIO is used to build high performance infrastructure for machine learning, analytics and various application data workloads.

3.1 Deployment

MinIO provides different configurations of hosts, nodes, and drives depending on the needs and infrastructure of the organization hosting the object storage service. There are three distinct types of deployment modes for MinIO: a) Standalone, b) Distributed, and c) Cloud Scale. In this work we will focus on the Standalone deployment of MinIO. To host multiple tenants on a single machine, we run one MinIO Server per tenant with a dedicated HTTPS port, configuration, and data directory.

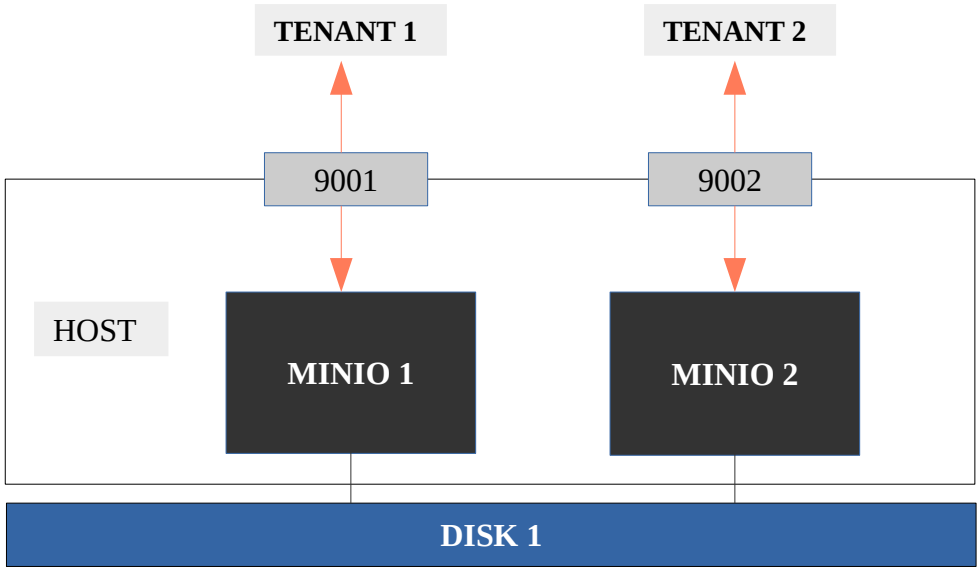

Figure 2: Two tenants on a single host with a single drive

In Figure 2 we illustrate a typical multi-tenant standalone deployment of MinIO on a single host with a single drive. In this example we have two tenants, each hosting a separate instance of MinIO on the same node. Each tenant has a dedicated port (port 9000 is usually used for standard MinIO deployment) , in order to forward HTTP requests to each MinIO instance. Given that we have successfully obtained the MinIO executable by compiling the source code, in order to host two tenants we use the following commands:

```
minio server --address :9001 /mnt/data/tenant1
minio server --address :9002 /mnt/data/tenant2
```

3.2 Accessing MinIO Server with TLS

In order to securely access MinIO server all communication between a client and a MinIO instance is protected via Transport Security Layer (TLS). The primary use of TLS is for encrypting the communication between applications and servers such as web browsers loading a website [8]. TLS accomplishes to enforce three fundamental security primitives between the communication of clients and MinIO:

- **Encryption:** hides the data being transferred from third parties.
- **Authentication:** ensures that parties exchanging information are who they claim to be.
- **Integrity:** verifies that the data has not been forged or tampered with.

To securely establish communication, we need to provide a private key and a public certificate that have been obtained from a certificate authority (CA). Alternatively, if the server is used for development purposed and not for actual production release, the hosting provider can generate a self-signed certificate using the openssl library [9]. First, we generate a private RSA key, 2048 bit long modulus, and store the key as *private.key*. As a next step we generate an SSL configuration file named openssl.conf stating information such as the location of the provider, domain name, IP address etc. By running the command:

```
openssl req -new -x509 -nodes -days 730 -key private.key -out public.crt -config openssl.conf
```

we eventually generate the certificate file named *public.crt*. The private key and the certificate signed using the private key, are stored under the MinIO configuration directory.

3.3 Encryption

As described in paragraph 3.2, the communication between client and MinIO server is securely established using TLS/HTTPS. Therefore, all data transmitted through the HTTP request headers is encrypted and integrity protected. However, when we successfully put an object into a bucket the data resides in plaintext and no form of encryption is applied by default, making the data publicly accessible to any client that gains access to the MinIO server. In order to further protect the privacy of our data MinIO supports two different types of server-side encryption (SSE):

- **SSE-C:** The MinIO server en/decrypts objects using a secret key provided by the S3 client as part of the HTTP request headers. SSE-C requires TLS/HTTPS.
- **SSE-S3:** The MinIO server en/decrypts an object with a secret key managed by a Key Management System (KMS).

MinIO uses a unique, randomly generated secret key per object known as Object Encryption Key (OEK). Neither the client-provided SSE-C key nor the KMS-managed key is directly used to en.decrypt an object. Instead, the OEK is stored as part of the object metadata next to the object in an encrypted form. To en/decrypt the OEK another secret key is needed also known as, Key Encryption Key (KEK). To summarize, for any encrypted object there exists three different keys:

- **OEK:** A secret and unique key used to encrypt the object, stored in an encrypted form as part of the object metadata and only loaded to RAM in plaintext during en/decrypting the object.
- **KEK:** A secret and unique key used to en/decrypt the OEK and never stored anywhere. It is (re-)generated whenever en/decrypting an object using an external secret key and public parameters.
- **EK:** An external secret key - either the SSE-C client-provided key**.**

For content encryption MinIO server used AES-256-GCM. Any secret key is 256 bits long. When applying SSE-C, the S3 clients need to send the secret key as part of the HTTP request. The secret key is never stored by the server however, it resides in RAM during en/decryption process. The client provided key is not required to be unique and multiple objects may be en/decrypted using the same secret key [10].

3.4 Threat Model

In our work, we focus on three kinds of principals. A **user** is someone who wants to store data online and may selectively expose the data to a **third party service** or another user of the system. The user is allowed to upload and store her data on MinIO which claims the role of a **cloud storage provider**. Each user has one storage provider, but potentially many third-parties (or users) which need access to the users data [11]. The user can upload her data in plaintext or in an encrypted form via user provided secret keys. By default client-server communication is encrypted and integrity protected using TLS/SSL. When applying SSE-C encryption user data resides on the MinIO server in an encrypted form. However, in case the storage server is compromised, during en/decrypting an object the OEK which is loaded to RAM, a malicious user (or third-party) which has legal access to the server, may launch cold boot attacks to retain the DRAM data by freezing the memory chip [12], thus gaining access to all objects which have been encrypted with the same OEK. To prevent data leakage by the execution of these kind of attacks, we propose a client encryption scheme where the data is formerly encrypted on the client side before uploading the data. Thus we do not leverage SSE-C encryption on the server side and never leak the secret key, since en/decrypting occurs on the client side and the keys are solely handled by the client. Using this encryption scheme, we can protect sensitive data leakage even if the communication with server is not protected with TLS since the data will already be sent in encrypted form without the HTTP header containing the secret key. However, we do not provide any security guarantees in case the client machine is compromised or when applying key rotation on the server.

4. Encryption Scheme

In this section we provide an overview to the encryption scheme we employed in order to encrypt the data that is sent to the untrusted storage provider.

4.1 Client Side Encryption

MinIO server is written in Golang, thus in order to support a client side encryption scheme were the data which is sent in an encrypted form to the storage provider we leveraged the encryption capabilities of Golang. Golang, is an open source programming language designed at Google. It's statically typed and produces compiled machine code binaries and is syntactically similar to C, but with memory safety, garbage collection, structural typing [13]. Figure 3 depicts the code used to encrypt the data before sending in to the untrusted storage server.

```
1.  /* export encrypt */
2.  func encrypt(fileToEncrypt string, fileName string, pathToKey string) *C.char{
3.
4.          encrypted_file := fileName + ".bin"
5.          plaintext, err := ioutil.ReadFile(fileToEncrypt)
6.
7.          /* Read key file content and convert it to string */
8.          content, err := ioutil.ReadFile(pathToKey)
9.          keyString := string(content)
10.
11.         /* Decode key in order to get the key as type byte[] */
12.         key, _ := hex.DecodeString(keyString)
13.
14.         /* Create a new Cipher Block from the key */
15.         block, err := aes.NewCipher(key)
16.
17.         /* Create a new GCM */
18.         gcm, err := cipher.NewGCM(block)
19.
20.         /* Never use more than 2^32 random nonces with a given key */
21.         nonce := make([]byte, gcm.NonceSize())
22.         if _, err := io.ReadFull(rand.Reader, nonce); err != nil {
23.             log.Fatal(err)
24.         }
25.         /* Encrypt the data using GCM Seal function */
26.         ciphertext := gcm.Seal(nonce, nonce, plaintext, nil)
27.
28.         /* Save back to file */
29.         err = ioutil.WriteFile(encrypted_file, ciphertext, 0777)
30.
31.         /* Returns a pointer to the encrypted file */
32.         return C.CString(encrypted_file)
33. }
```

Figure 3: AES-256 GCM Encryption in Golang.

In order to encrypt the data, we have written a function in Go which leverages AES-256 GCM encryption. AES-256 GCM is a mode of operation for symmetric-key cryptographic block ciphers similar to the standard counter mode [14]. This encryption module takes as input a file to encrypt (which can be any type of file type since its

read as a byte sequence), the raw file name without the path and the file ending, and a symmetric key encoded in base64. The encrypted file is stored to the file system with all permissions granted (mode 777). Finally the functions returns a C string, which is essentially a pointer to the encrypted file. Go supports building the code and exporting C shared compiled libraries which can be distributed, installed and linked to every other application in other programming languages that can reference C shared libraries (C++, Python, Javascript, etc). In order to export the C shared library for en/decryption functionalities which reside in source file *libencrypt.go* we used the following command:

```
go build -buildmode=c-shared -o libencrypt.so libencrypt.go
```

The output of the above mentioned command is: a) *libencrypt.so,* the C shared libreay, and b) *libencrypt.h*, the C header file. Figure 4 presents a sample Python code used for encrypting objects using the C shared library:

```
1.  /* Loading C shared encryption library */
2.  lib = cdll.LoadLibrary("libencrypt.so")
3.
4.  /* Structure that handles type convert between modules */
5.  class go_string(Structure):
6.  _fields_ = [
7.  ("p", c_char_p),
8.  ("n", c_int)]
9.
10. /* Encryption function wrapper */
11. def encrypt(file_path, file_name, AES_KEY):
12.     lib.encrypt.restype = c_char_p
13.     fp = go_string(c_char_p(file_path.encode('utf-8')), len(file_path))
14.     key = go_string(c_char_p(AES_KEY.encode('utf-8')), len(AES_KEY))
15.     file_name = go_string(c_char_p(file_name.encode('utf-8')), len(file_name))
16.     encrypted_file_name = lib.encrypt(fp, file_name, key)
17.
18.     /* Returns the name of the encrypted file */
19.     return encrypted_file_name
```

Figure 4: Python code leveraging C shared library encryption module.

4.2 Server Side Encryption

The default encryption scheme by MinIO server is Server-Side Encryption (SSE) in order to securely encrypt the uploaded objects within the object storage itself. In our work we will implement SSE-C with client provided keys managed solely by the client. In contrast to the forementioned client side encryption scheme,

where the presence of TLS was not obligatory since the data was uploaded in an encrypted form, SSE-C requires a valid SSL certificate and a client provided key which is attached to HTTP request headers.

```
1.  /*  Encodes key and returns SSE configuration */
2.  def sse_encryption(key):
3.
4.      f = open(key, "r")
5.      key_str = f.read()
6.      key_str = key_str.replace('\n', '')
7.      key = key_str.encode('ascii')
8.      SSE = SseCustomerKey(key)
9.
10.     return (SSE)
11.
12. /* Custom HTTP client inquiring valid SSL certificate */
13. httpClient = urllib3.PoolManager(
14.             timeout=urllib3.Timeout.DEFAULT_TIMEOUT,
15.                 cert_reqs='CERT_REQUIRED',
16.                 ca_certs=PUBLIC_CERTIFICATE,
17.                 retries=urllib3.Retry(
18.                     total=5,
19.                     backoff_factor=0.2,
20.                     status_forcelist=[500, 502, 503, 504]
21.                 )
22.         )
23. /* MinIO Client initialized with access information */
24. client = Minio(MINIO_URL,
25.                 access_key='minio',
26.                 secret_key='minio123',
27.                 secure=True,
28.                 http_client=httpClient)
29.
30. /* SSE Customer provided key encryption */
31. SSE = sse_encryption(key_path)
32.
33. /* Clients putting object to server bucket with SSE encryption */
34. client.fput_object( BUCKET_NAME, file_name, file_path, sse=SSE )
```

Figure 5: Python code for an uploading object using SSE-C encryption.

Figure 5 illustrates the Python code responsible for inquiring an SSE client provided key encryption. Function *sse_encryption( )* is responsible for encoding the input key and generating the corresponding SSE configuration. Lines 13-28 describe the initialization of the HTTP (custom) and MinIO clients provided with the server credentials and the public certificate for executing the TLS/SSL protocol.

4.3 Key Rotation

Another important aspect that we examine in this work, is how to revoke access to a third-party (or user) that gained access to the unique symmetric en/decryption key of an object that is already stored on the MinIO server. Even if the object is encrypted, a curious user who is not the owner of the file, may have access to the object and decrypt it as he was granted permission to download the object e.g., the third-party may have been formerly listed in the Access Control List (ACL) of the object. In order to prevent revoke access to a third-party,

a possible solution would be to re-encrypt the object on the storage provider. Let K_{obj} be the symmetric key of an uploaded object on a MinIO server instance. Re-encrypting the object would require to decrypt the object with its original key K_{obj} and re-encrypt the object with a new key K'_{obj}. The secure distribution of the keys is beyond the scope of this work. To the best of our knowledge, MinIO and other related open-source object storage platforms do not provide any re-encryption capabilities on the server side. However, S3 clients can change the client-provided key of an existing object by performing an S3 COPY operation where the coy source and destination are equal. The HTTP COPY request headers must contain the current key K_{obj} and the new client key K'_{obj}. Such a special COPY request is know as S3 SSE-C key rotation.

```
1.  /* Source object customer provided key */
2.  SSE_SRC = sse_encryption(OLD_AES_KEY)
3.
4.  /* Destination Object SSE Customer provided key encryption */
5.  SSE_DST = sse_encryption(NEW_AES_KEY)
6.
7.  /* Copy the object to the same bucket using a different key.
8.   * Object does not exit the server side using this method.
9.   */
10. result = client.copy_object(
11.       BUCKET_NAME,
12.       file_name,
13.       CopySource(BUCKET_NAME, file_name, ssec=SSE_SRC),
14.       sse=SSE_DST,
15. )
```

Figure 6: Python code for executing S3 SSE-C key rotation.

In Figure 6, we list the Python code used to implement the S3 COPY operation which results in key rotation of the specified object. The client initialization and SSE configuration method are not included since they are stated in Figure 5. The client (owner of the file) generates two distinct SSE configurations (lines 1-5) using the old encryption key K_{obj} and the new encryption key K'_{obj} which she generates and handles privately. Lines 11-14 specify the contents of the HTTP request headers including the SSE configurations the selected object and the bucket in which the object resides. On the server side, the MinIO server decrypts the OEK which is part of theobject metadata using the KEK derived from K_{obj}. The server receives the new key K'_{obj} and derives a new KEK with it. Using the newly generated KEK the server re-encrypts the OEK with it. As a final step, the freshly encrypted OEK data key is stored as port of the object metadata.

5. Evaluation

In this section we evaluate the performance of the encryption schemes we employed in our standalone deployment of the MinIO object storage server.

5.1 Experimental Setup

In order to evaluate the scaling performance of: a) our client side encryption scheme, b) the MinIO SSE-C encryption, and c) the SSE COPY request leveraged for key rotation, we deploy a standalone version of MinIO built from source. The standalone deployment consists of one node equipped with Intel (R) Core (TM) i5-4590 at 3.30GHz with 4 cores, 7.7 GB of RAM and a 500 GB hard disk drive. The clients and the MinIO server are connected through the the loopback interface since both are located on the same node. Our system runs the latest version of MinIO built from source and we use Go version go1.15.6 linux/amd64. The operating system on the node is Debian 10 buster 64-bit Linux distribution 4.19.0-13-amd64. The private key and the public certificate reside within the MinIO configuration directory and were generated using OpenSSL version 1.1.1d. Symmetric keys for object AES-256 object encryption are 32 byte long and derived by a random generator implemented in Go leveraging module *crypto/rand*.

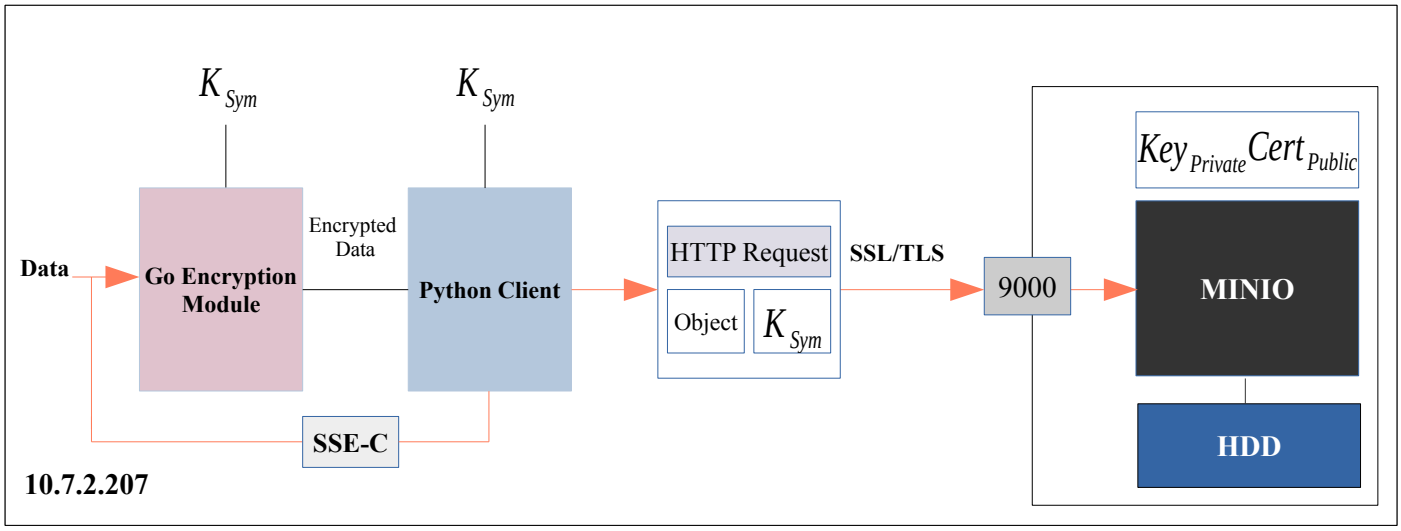

Figure 7: MinIO Standalone Deployment

Figure 7 illustrates the Standalone MinIO server setup on a single server with IP: 10.7.2.207. The data to encrypt resides on the file system and is encrypted using the the *Go Encryption Module*. The symmetric key K_{Sym} for encryption is already generated by a Golang random key generator module which we do not depict here. The *Encrypted Data* and K_{Sym} are propagated to the *Python Client* which generates an HTTP Request containing the encrypted data object and optionally K_{Sym}. When applying SSE-C encryption we bypass the *Go Encryption Module* and propagate the plaintext *Data* to the client. The client communicates via SSL/TLS and

port 9000 with the MinIO server, which provides its public certificate $Cert_{Public}$ signed by its private key $Key_{Private}$.

5.2 Performance

We tested the performance of Apache Spark Streaming and Apache Flink using the benchmark application of WordCount which was described in earlier sections under different cluster setups, scaling from 8 to 32 cores (one to four worker nodes). In our tests the mean value of the data sent to the cluster per second was 50K records/second. To collect performance metrics we used the tool Spark Measure [8] for Spark and for Flink its Monitoring REST API [12].

Figure 8:

Figure 9:

Figure 7 illustrates the throughput attained by the frameworks. We can see that at a cluster of only one node (8-cores) Spark-Streaming attains a slightly higher throughput than Flink, processing roughly 3.4 Mb of data. Both frameworks scale almost linearly to 4 nodes (32 cores) with Flink outperforming Spark-Streaming and reaching a maximum throughput of ~ 11 Mb/s, where Spark attains a throughput of ~ 10 Mb/s.

Figure 8 reports the time spent on CPU usage of both frameworks at the given workload. The total execution time of the benchmark was 5 minutes (300 seconds). At cluster setup of one node Flink fully utilizes the CPU as the CPU Time is almost 300 seconds where Spark makes use of the CPU resource for about 260 seconds. While scaling to larger clusters both frameworks spend less time on CPU due to network latency. This network latency lies on the fact that by adding more worker nodes to the cluster, time is spent on the communication and coordination of the nodes by the Master node. In that aspect, Flink makes better use of the cores assigned to the cluster at all the given cluster setups.

Figure 9: Network utilization percentage of the data analytic frameworks.

Figure 9 illustrates the network utilization percentage of the frameworks at the given cluster setups. At a cluster of one node (8 cores), both frameworks utilize the network approximately at 10% given that not much time is spent on communication using only one worker node. At a two-worker node setup, Spark spikes to ~ 32% of network utilization spending more time in communication where Flink sustains a utilization of roughly 11%. Scaling to a cluster of four-worker nodes, the network utilization of Spark reaches to ~ 42% spending almost half of the execution time in network communication and Flink about 32%.

6 Conclusion

7. References

[1] MinIO, High Performance, Kubernetes Native Object Storage https://min.io/

[2] Amazon Simple Storage Service https://aws.amazon.com/s3/

[3]Neal Ekker “File and object storage for dummies” A Willey Brand.

[4] Nagapramod Mandagere,Ramani Routray, Yang Song: “Cloud Object Storage Based Continuous data Protection(CCDP) ”, 2015 IEEE International

[5] Object Storage https://www.ibm.com/cloud/learn/object-storage

[6] Amazon Simple Storage Service, Developer Guide https://docs.aws.amazon.com/AmazonS3/latest/dev/

[7] MinIO Multi-Tenant Deployment https://docs.min.io/docs/multi-tenant-minio-deployment-guide.html

[8] Transport Layer Security https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/

[9] Securely Accessing MinIO https://docs.min.io/docs/how-to-secure-access-to-minio-server-with-tls#generate-a-self-signed-certificate

[10] MinIO Security Overview https://docs.min.io/docs/minio-security-overview.html

[11] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. 2017. Sieve: actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (*Middleware '17*). Association for Computing Machinery, New York, NY, USA, 14–27.

[12] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019* (*EuroSys '19*). Association for Computing Machinery, New York, NY, USA, Article 14, 1–15.

[13] The Go Programming Language https://golang.org/doc/

[14] Golang Package Crypto https://golang.org/src/crypto/