Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Karl Petter E. Aubert
Vidar Michaelsen

Viability of progressive decryption of large media files using chunk-based storage

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2023

NTNU
Norwegian University of
Science and Technology

Karl Petter E. Aubert
Vidar Michaelsen

Viability of progressive decryption of large media files using chunk-based storage

Master's thesis in Informatics
Supervisor: Svein Erik Bratsberg
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

This thesis aims to find out whether or not a document-oriented database employing a chunk-based storage strategy has satisfactory read performance compared to traditional NTFS file server-based approaches when accessing encrypted data. The identified benefits of using chunk-based storage is that random read access decryption could be sped up significantly, and this would allow for media streaming of encrypted content. We implemented three solutions, two of which were solely on the disk, while the last used a proprietary document-oriented database. The first method decrypts entire files at a time from the disk. The second splits files into chunks such that they can be decrypted regardless of each other. The third is similar to the chunk-based approach but stores the chunks in a document database. The first solution performs well for smaller file sizes but degrades as files eat up the main memory. The latter two solutions use less memory and execute faster for most file sizes. The database approach is somewhat slower due to the overhead associated with data fetching, but benefits such as support for transactions and having a single source of truth make it a competitive and viable option.

Oppsummering

Denne oppgaven har som mål å finne ut om en dokumentorientert database som bruker en lagringsstrategi basert på filoppdeling vil ha tilfredsstillende leseytelse sammenlignet med tradisjonelle NTFS-filserverbaserte løsninger ved tilgang av krypterte data. Det ble observert fordeler ved bruk av filoppdeling under lesing, blant annet raskere lesehastighet og lavere minnebruk. Vi implementerte tre løsninger for å sammenlikne ytelsen. De to første tok kun i bruk lagring på disk, og den siste tok i bruk en proprietær dokumentorientert database. Den første løsningen krypterer og dekrypterer hele filer fra disk. Den andre tar i bruk filoppdeling som gjør at deler av filen kan dekrypteres uavhengig av andre deler. Den siste løsningen tar i bruk en dokumentorientert database for å lagre delene av filen, i stedet for å lagre rett på disk. Den første, naive løsningen, har god ytelse for små filer, men blir verre når filstørrelsen øker og RAM-en blir brukt opp. De to siste løsningene bruker mindre minne, og gjennomstrømningen er nesten lineær, uavhengig av filstørrelse. De er også raskere for de fleste filstørrelser. Databasetilnærmingen går noe saktere på grunn av ekstra arbeid relatert til å hente ut data fra databasen framfor fra disk. Den kan likevel ha fordeler i forbindelse med databasetransaksjoner og at den har én enkelt sannhetskilde.

Contents

1 Introduction **1**

1.1 Research question 2

1.2 Structure 2

2 Background **4**

2.1 Cryptography 4

2.1.1 Encryption overview 4

2.1.2 Encryption modes 7

2.1.3 Cipher block chaining 9

2.1.4 Encryption performance 9

2.2 Progressive video download 11

2.2.1 Requirements 12

2.2.2 Bitrate 12

2.3 State of file storage 13

2.3.1 File system 14

2.3.2 File server 14

2.3.3 Relational database management system 15

2.3.4 Chunk storage 15

2.3.5 External solutions 16

3 Methodology **18**

3.1 Defining solutions 18

3.1.1 *Disk* solution 18

3.1.2 *Chunk* solution 19

3.1.3 *DBC* solution 20

3.2 Performance indicators 21

3.2.1 Time to first byte 21

3.2.2 Throughput . . . 21
3.2.3 Memory usage . . . 22
3.2.4 Other performance indicators . . . 22
3.3 Test bed . . . 24
3.3.1 Choice of programming language . . . 24
3.3.2 Software breakdown . . . 25
3.3.3 Hardware breakdown . . . 27
3.3.4 Test parameters . . . 28
3.4 Benchmarking solutions . . . 29
3.4.1 Ensuring validity of the tests . . . 29
3.4.2 Benchmarking application . . . 31

4 Results and Analysis **36**
4.1 Encryption . . . 36
4.2 Decryption . . . 40
4.2.1 Dry run . . . 43
4.2.2 Time to first byte . . . 44
4.3 Analysis . . . 45

5 Conclusion **47**
5.1 Future work . . . 48
5.1.1 Measurement of CPU times . . . 48
5.1.2 Chunk storage in RDBMS . . . 48
5.1.3 Break-even points for chunk sizes . . . 48

List of Figures

2.1 Diagram of symmetric encryption [11]. 6
2.2 Diagrams depicting CBC encryption and decryption [21][22]. 10
2.3 Structure of index file 16
2.4 Structure of chunk 16

3.1 Relationship diagram of the application, highlighting the most important aspects. 32
3.2 Flow of the chunk-wise encryption algorithm. 34

4.1 Average encryption time (logarithmic scale) 38
4.2 Average encryption throughput 38
4.3 Memory peak during encryption (logarithmic scale) 39
4.4 Average decryption time (logarithmic scale) 41
4.5 Average decryption throughput 41
4.6 Memory peak during decryption (logarithmic scale) 42
4.7 DBC dry run compared to average (seconds) 43
4.8 Time to first byte 44

List of Tables

2.1 Modes of operation and their support for random read access 9

3.1 Software and libraries used. 27
3.2 Hardware components of test bench. 28

4.1 Encryption time in seconds 37
4.2 MB/s encrypted 37
4.3 Encryption memory peak 39
4.4 Decryption time in seconds 40
4.5 MB/s decrypted 40
4.6 Memory peak during decryption 42
4.7 DBC dry run time compared to average (percentage) 43
4.8 Time to first byte 44

Chapter 1

Introduction

Multimedia streaming plays a big role in internet traffic. While many streaming services are centered around public content, which requires few security measures, others are centered around content that should be kept more private. For example, a company may wish to keep records of their online meetings as video so that employees can review them later through video on demand (VOD), skipping to relevant parts of the video as they wish. To ensure that their content is not accessed by unauthorized entities and that privacy laws are upheld, they can encrypt their videos at rest using private key encryption. The issue with this method is the latency incurred by having to decrypt these potentially large media files when requesting to view a video, especially when multiple users are requesting different videos and overtaxing the system.

One approach to mitigate this issue is to stream the files from disk, partially decrypting them on the fly. This approach is usually combined with some form of a relational database that holds metadata associated with the files. Since the metadata and the actual files are separated, this approach could introduce problems with consistency as there is no single source of truth if files were to be deleted on one end and not the other.

We identified a potential solution to such cases in which metadata and file contents are stored within the same part of the system. The proposed solution uses a proprietary document-oriented database with built-in support for subdividing files into smaller pieces (chunks). To assess the feasibility of such a system, it needs to be able to handle large files while being highly performant in terms of execution speed and resource usage.

1.1 Research question

Using the document-oriented solution, we present the following research question: How might a chunk-based document database assist with the process of streaming of data that is encrypted at rest?

To cater to this, we need to do the following:

1. Implement a simple encryption method that works on entire files on disk.
2. Develop a method to encrypt and decrypt files in chunks.
3. Implement this method both for disk and for a document database.
4. Benchmark and compare the two disk approaches against the database approach.
5. Identify the benefits and shortcomings of using the database chunk method compared to the other tested methods.

After going through these steps, we will discuss the results, possible points of improvement, and how the research can be furthered.

1.2 Structure

The report is structured to give a rough overview of components surrounding file storage before explaining the specific domain more thoroughly. It is divided into the following sections:

- **Chapter 2 — Background:** In this chapter, we provide a concise overview of encryption, video streaming, and common file storage methods as they relate to our research question. We aim to cover the fundamental principles of these topics without compromising clarity and understanding.
- **Chapter 3 — Methodology:** Here, we cover the inner workings of the solutions and how they differ from each other. We also present performance indicators to evaluate their effectiveness and provide a clear explanation of the test bed used for assessment.

- **Chapter 4 — Results and Analysis:** The testing results are presented, visually representing the data through informative graphs. We then analyze these results, aiming to decipher their implications and gain valuable insights into their significance.
- **Chapter 5 — Conclusion:** The last chapter recaps the thesis and aims to draw conclusions from the findings and how research on the topic can proceed.

The Introduction, Background, and Methodology chapters are based on our preparatory thesis.

Chapter 2

Background

This chapter provides an overview of the relevant fields which are required to detail the methods and their implementation discussed in later sections. Guided by the research question, we will focus discussion only on the parts which are deemed relevant.

2.1 Cryptography

Cryptography is the field of secrets, dedicated to exploring how data can be kept private both in storage and in messages through the use of complex mathematics and secret keys. This section will outline relevant terms and algorithms. Our main concerns are *performance*, *confidentiality*, and *random access decryption*. Confidentiality means that data can only be accessed by authorized parties [1], while random access decryption means being able to decrypt any section of an encrypted file without having to decrypt the entire file.

2.1.1 Encryption overview

In modern terms, cryptography is often used to obfuscate files such that they are confidential [2]. Obfuscating a file is known as *encryption*, turning *plaintext* into *ciphertext* [3]. Plaintext is simply a regular file which can be opened and read as normal. The ciphertext is meaningless unless it is turned back into plaintext through *decryption*, the inverse process of encryption [4]. Encryption is also sometimes used to refer to the entire process of securing data, including the decryption process. A common example of this is the term *encryption algorithm*.

An encryption algorithm is the central component of encryption, and each algorithm is a specific set of mathematical steps to take in order to encrypt and decrypt files [5]. Encryption algorithms use keys to ensure data is kept secure. A key is simply a randomly generated alphanumeric text of a certain length which is used as a parameter when encrypting files [6]. To decrypt files encrypted with such a key, a corresponding key is required. The exception to this is *cryptanalysis*, which is the process of exploiting the weaknesses of encryption algorithms, allowing access to the underlying data without the key [7]. This is also known as *cracking*.

In contrast to many other fields, which do not always concern themselves with potential adversarial efforts to attack their creations, the concept of confidentiality inherently assumes the existence of unauthorized parties who must not access encrypted data. While cracking an encryption algorithm takes a lot of work, there is much to gain by being able to access someone else's confidential data. Some algorithms have been cracked due to a mathematical weakness, while others have been left unusable due to advances in computing power and how easy brute-force attacks have become on algorithms using shorter keys. Encryption algorithms are designed with cryptanalysis in mind and are designed such that cracking them should ideally require a level of time and effort that is, in practice, infeasible. Measuring the security of an algorithm is a complex task, as new exploits can be found at any time. A few of the more objective measures of security are how resistant the algorithm is to known attacks, and the length of the key.

The main *encryption schemes* are either *asymmetric* or *symmetric* [8] [9]. Asymmetric encryption uses a public key for encryption and a private key for decryption. This allows users to encrypt messages that only the recipient can access using their private key without exchanging any private information beforehand. In symmetric encryption, a single key is used for both operations. Asymmetric encryption generally requires more complicated mathematical operations than symmetric encryption, making it less performant and less straightforward to implement [10]. As all encryption and decryption in our solutions will occur on the same system, only symmetric encryption is relevant to our thesis. An illustration of how symmetric encryption functions can be seen in Figure 2.1.

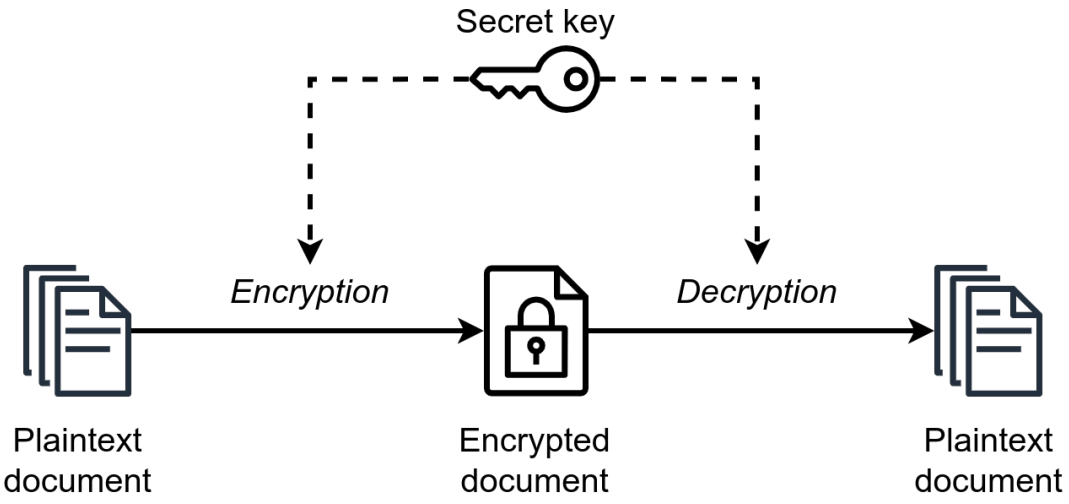


Figure 2.1: Diagram of symmetric encryption [11].

Symmetric key encryption can be done using either a *block cipher* or a *stream cipher* [12] [13]. A block cipher splits input plaintext into the eponymous fixed-size blocks before encrypting them, adding padding to the last block to ensure the blocks are all the same size. Many different *encryption modes* or *modes of operation* exist for block ciphers, which differ in safety, performance, resource utilization, and some other key attributes. A stream cipher uses a key to generate a *keystream*, a pseudorandom string that is equal in size to a target plaintext to encrypt [14]. Keystreams do not have to be fully generated at once, but they do have to be sequentially generated. As such, decrypting the last half of a file requires generating the entire keystream. This means that although stream ciphers can be faster than block ciphers while requiring less memory and complexity, our constraint of requiring constant-time random access decryption requires the use of a block cipher.

The relevant block cipher encryption algorithms and a short explanation of each follow:

- *Advanced Encryption Standard [15]*: Known as AES, based on Rijndael. It is the industry standard for encryption, chosen and supported by the National Institute for Standards and Technology (NIST). Many modern processors have

hardware support for this algorithm with the AES New Instructions (AES-NI) instruction set, which provides an inherent performance boost to this type of encryption. Supports key lengths of 128, 192 and 256 bits.

- *Data Encryption Standard [16]*: DES, for short, was the predecessor of AES. It is known to be insecure due to its short key length, which is why 3DES was invented. 3DES is a variant of DES that applies the same algorithm to each block three times in total instead of just once. This can effectively triple the key length without modifying the underlying DES algorithm. Has a key length of 56 bits, with 112 or 168 bits for 3DES.
- *Twofish [17]*: One of the other finalists in the selection to become AES, Twofish was passed over due to the winning algorithm, Rijndael, possessing what was deemed to be a superior balance of performance, security, and simplicity. As it was designed for the same purpose as Rijndael, it also supports keys with lengths 128, 192, and 256 bits.

Due to AES's position as the industry standard, it was chosen as the basis for our development.

2.1.2 Encryption modes

A block cipher only does work on single blocks at a time, which AES defines as 16 bytes in length. When combined with a mode of operation, the cipher can encrypt files of arbitrary size. Although modern modes include *integrity* guarantees, they are out of the scope of this thesis. Integrity means ensuring that encrypted data has not been altered in an unauthorized manner [18]. This is mostly a concern in data communication and might be a viable expansion of scope for future work. We will only consider the simpler, confidentiality-only modes described by NIST in 2001 [19].

The five classic modes of operation, according to NIST, are Electronic Codebook (ECB), Cipher Block-chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR). Whether or not a mode of operation allows for random read access can be seen by what each block requires as input to be decrypted. Com-

mon for all modes of operation is that each block requires both the ciphertext of the relevant block and the key, and several also require a pseudorandom initialization vector (IV) as a seed [20]. Any mode which requires input from the decryption process of a preceding block, such as its plaintext or any result of its decryption step, is locked to linear decryption and is, as such, not relevant for our purposes.

ECB is a highly simplistic mode that requires no additional information to decrypt any block. This means all blocks can be decrypted independently, but it also means that any block with identical plaintext is encrypted identically if using the same key. This allows attackers to sense patterns in the ciphertext which match patterns in the plaintext. Due to this, ECB is widely considered insecure for most purposes despite otherwise passing the criteria for our solution.

Both CBC and CFB only require the ciphertext of the preceding block, meaning it is possible to start decrypting at any block. Retrieving a single block will still require reading the ciphertext of another block, meaning this method is not the most memory efficient, but this is negligible as blocks are often retrieved in large batches.

OFB requires each block's input to come from the decryption process of the preceding block, making it ineligible for random read access and thus not relevant to our solution.

CTR is very similar to OFB, but where the input to OFB comes from the IV being encrypted once for each block, the input to CTR is simply a unique number for each block, commonly a simple incrementing counter and a *nonce*. Nonce is short for "number used once" and is similar to the IV of other modes. This means each block can be decrypted independently of all others as long as the initial counter value and the sequence number of the block are known.

A summary of the modes supporting random read access can be seen in Table 2.1. The viable modes for this thesis are CBC, CFB, and CTR. Due to prior experience with CBC and the relative ease of implementing it, it was deemed the most suitable for our requirements despite CTR being slightly more memory efficient.

Mode	Additional input for each block	Random read access
ECB	None	Yes
CBC	Preceding ciphertext	Yes
CFB	Preceding ciphertext	Yes
OFB	Preceding encrypted IV	No
CTR	Nonce and counter	Yes

Table 2.1: Modes of operation and their support for random read access

2.1.3 Cipher block chaining

Cipher block chaining works by splitting the input plaintext into 16-byte blocks. Each block of plaintext is XORed[1] with a vector before being encrypted with the key. For the first block, this vector is the randomly generated initialization vector. For every subsequent block, the preceding block's ciphertext is used as the vector. The decryption process is similar but in reverse. The key is first used to decrypt the ciphertext, and the result of this operation is then XORed with the same vector the block was initially encrypted with.

A diagram of the process of cipher block chaining encryption and decryption can be seen in Figure 2.2.

2.1.4 Encryption performance

It is worth noting the factors which impact the speed at which encryption and decryption can take place, as performance is an important aspect of this study. The information collected here informs the analysis and conclusions drawn from the results.

As stated earlier, symmetric encryption vastly outperforms asymmetric encryption due to the added mathematical complexity secure asymmetric encryption requires. Commonly, a hybrid approach is used to transfer large encrypted files. This approach involves using asymmetric encryption to exchange a symmetric key which is then used for the actual data to be transferred [23]. This method can give the benefits of both

[1]Exclusive or ($\oplus$). A logical operation on two sets, resulting in a "truthy" value *only* for differing values. For example, $0011 \oplus 0101$ gives 0110.

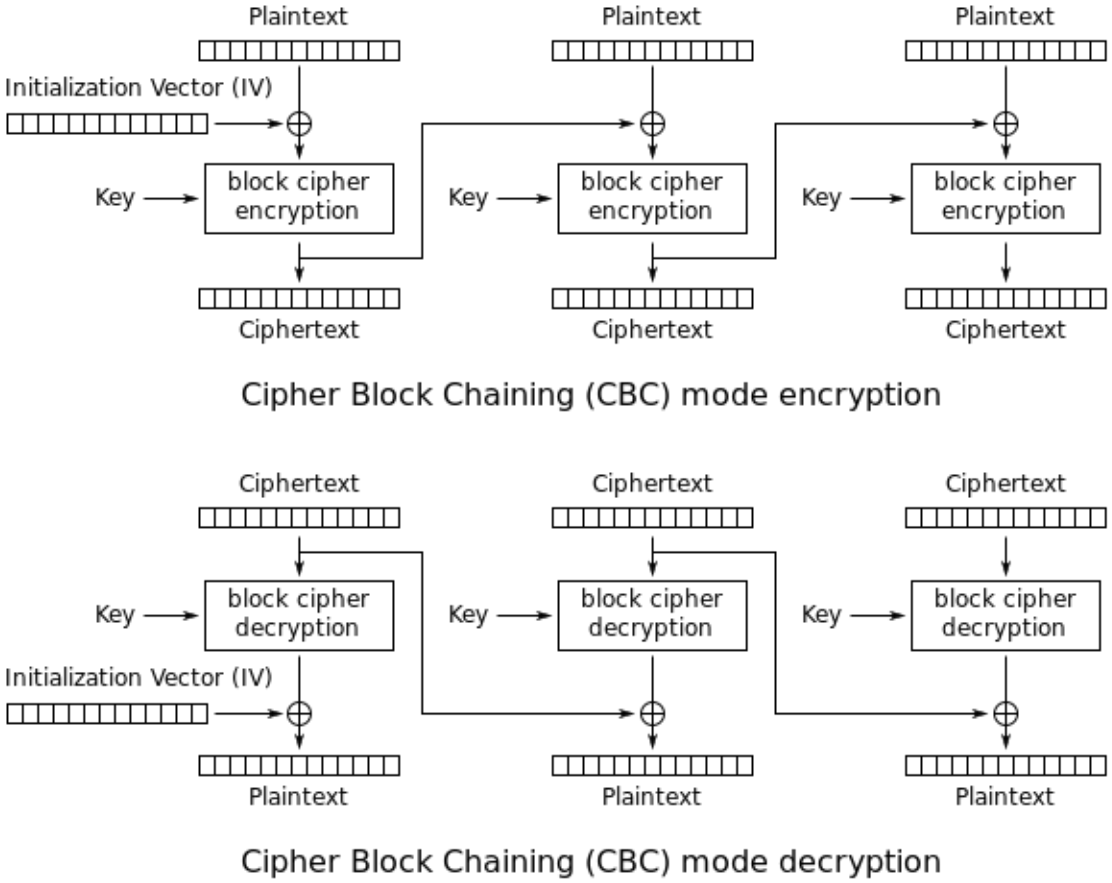


Figure 2.2: Diagrams depicting CBC encryption and decryption [21][22].

asymmetric and symmetric encryption when communicating through unsafe networks and could be relevant to the solution we propose in the future.

The first and perhaps most obvious factor impacting the encryption performance of a certain file is the size of the file itself. As each block cipher encryption algorithm has to do its work on every block of data, it stands to reason that multiplying the number of blocks also multiplies the processor cycles required to complete the task. This relationship is typically linear, meaning doubling the number of blocks means doubling the amount of time taken. To speed a given algorithm up, processors have to start becoming meaningfully faster again, or parallel computation has to be utilized. All three viable modes of operation for this thesis allow for parallel decryption, but

only CTR allows for parallel encryption.

The length of the key is also an important factor. Longer keys may impact encryption and decryption time, but certain algorithms are designed such that increasing key length does not dramatically increase encryption time. For AES, increasing the key increases the time spent per block somewhat, but this will not be measured in our thesis.

2.2 Progressive video download

The concept of progressive download is simple, although the execution can at times be complex. A common use case for progressive download is websites with large, high-resolution images. With this technique, images can start being rendered before being fully downloaded [24]. This allows users to start interacting with and viewing images in a more seamless manner. Progressive download can work for a broad selection of media types, including text, images, audio, and video, among others. An important caveat is that not all file formats support progressive download, such as formats that include metadata at the end of their files.

Progressive video download is mostly used for streaming. Streaming of video mainly refers to receiving and displaying video linearly, with a very limited (if any) buffer to hold the next frames. This can be likened to a TV receiving a live signal, with no ability to move forwards or backwards in the stream. Many modern streaming services offer video on demand and make use of progressive downloads. Progressive video download is a method of storing downloaded stream data in a temporary directory, allowing users to jump to previously downloaded sections of video. These services also often implement *seeking*, letting users skip to a point in the video they wish to download or view. When discussing video streaming in this thesis, we are referring to progressive download with seeking.

2.2.1 Requirements

In order for a file format to support progressive download, it has to be *streamable*. A streamable format has to play without issue even if the file is only partially downloaded. Not every video file format is streamable due to the requirements being very specific. The requirements are the following three criteria:

- *Metadata at the beginning*: Metadata means information about information. For a video file, this entails critical information needed for video playback such as duration, video resolution, and which *codec* is used [25]. The codec is used to *encode* video to reduce file size, and different codecs have different requirements for their playback environment.
- *Interleaved data*: The video format has to put the video and audio on the same track, interleaving them in small chunks so that video playback can begin as soon as possible. If these chunks are one second of video and audio, then each pair is a self-contained second. If the tracks were separate, it could be possible to desynchronize the downloads such that one track is waiting for the other, causing stuttering.
- *Data loss resilience*: For streaming, playback of an incomplete file is the default. If any segment of the video is missing. it should either be skipped or automatically paused until the next chunk is downloaded.

Many video file formats, especially those created specifically for web content and video streaming, such as MP4 and WebM, are streamable. While other file formats support streaming, finding, testing, and listing each of them is out of the scope of this thesis.

Another important concept in progressive video download is *bitrate*.

2.2.2 Bitrate

Bitrate is a measure of how many bits are required to convey a second of a continuous medium, such as audio or video. It is expressed as *bit/s* or *bps*, always in decimal units. While bitrate is a common measure of quality, two video files with equal

size, complexity, and bitrate could have vastly different qualities depending on the compression technique used. Better encoding can lead to higher quality at equal bitrates, so it is not a perfect measure.

A media file consisting of 10 seconds at a bitrate of $10Kbps$ would equal $100Kb$ of content. A typical uncompressed video with 8 bits per color channel, a full HD resolution ($1920 * 1080$) delivering 30 frames per second would have a bitrate of $24 * 1920 * 1080 * 30 = 1.49Gbps$. This can be massively reduced by encoding without reducing video quality much, if at all, so raw video is seldom transferred over the internet.

The number of bits needed to convey a frame or a series of frames can differ widely when encoded. Fewer bits are needed when the color of the frame is uniform or when a frame is similar to the previous frame, and more bits are needed if a frame is hard to simplify or the changes from frame to frame are drastic or unpredictable. This is sometimes referred to as *entropy*[2].

Encoding an entire video to the same bitrate would be using *constant bitrate*. In contrast, a *variable bitrate* entails compressing less complex frames into fewer bits and using more bits for frames of higher complexity. Constant bitrate allows for more predictable network load and buffer sizes while using a variable bitrate lets the file be encoded to a smaller size, with more detail in 'noisy' frames. Live streaming benefits greatly from the network advantages provided by a constant bitrate. Our solution does not require a specific encoding type, but variable bitrate might impact the system's perceived performance due to buffering. Video bitrates can vary wildly, but common bitrates for video streaming lie between 1Mbps and 300Mbps [26].

2.3 State of file storage

There are several approaches to storing media files, both in terms of *how* and *where* they are stored. Considerations must be made in order to know what solutions are best suited for a given application. Here we will discuss some of the different types

[2]Mostly relevant for finding lower bound of lossless compression and not relevant for our case.

of file storage.

2.3.1 File system

The term *file system* can be described in many ways [27][28], and their implementations can differ heavily. The Linux System Administrators Guide explains the following: "A *filesystem* is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk." An operating system (OS), like the Linux distribution Ubuntu, can use one of many file systems; ext3, ext4, and XFS, to name a few.

Popular file systems include New Technology File System (NTFS), which comes as default for machines running modern Windows versions, and ext4, which is the default for most Linux distributions. Variations between them include the method of indexing files and the metadata associated with the files [29]. Using different file systems for the same application can cause compatibility issues. This can be caused by different naming conventions (e.g., not allowing backslashes), file size limitations, and more.

Even though the file system is responsible for the connection of files between the hardware and OS, the choice of OS is mainly what governs what file system is used. We do not consider the choice of a file system to be particularly relevant for this study, but it can affect performance; the same goes for OS. It is worth noting that applications (including database software) may use the underlying OS' implementation to store files. An example is the *Filestream* data type found in SQL Server, which uses the file system to store unstructured data [30].

2.3.2 File server

A file server is a network-attached computer used to store files [31]. Corporations can use file servers to store documents, host media files for websites, and more. This usually allows easy access through remote desktop software, shared directories, and a secure shell.

One approach to using file servers for hosting media data is to use it with a relational

database. The database stores metadata (such as the file path) related to the files stored on the file server, and the file server contains the actual files. This approach allows clients to easily access file information without needing to retrieve the entire file. A drawback, however, is that it is not necessarily easy to ensure consistency between the database and the file server.

2.3.3 Relational database management system

Relational database management systems (RDBMS) are applications that provide functionality for storing relational data using SQL [32]. A relational database contains tables, which in turn contain rows of data. Each row consists of one or more attributes. A data type is connected to each attribute, for instance, *int* for integers, or *nvarchar(n)* for characters of max length n. The data types may vary between different systems.

Most RDBMSs include attribute types that support unstructured data. The most well-known is the *binary large object* (BLOB). A BLOB is not restricted to certain file types and accepts any form of binary data so long as it conforms to the column definition (e.g., max size). Using an RDBMS can negate the problem of ensuring consistency between the two platforms. As mentioned in Subsection 2.3.1, there may also be file types that adhere to the underlying file system.

2.3.4 Chunk storage

A commonly used document-oriented database supports what we have termed *chunk storage*. Document-oriented databases store documents in collections, where documents (often in a JSON[3] structure) are stored in collections [33]. Documents inserted into a collection can be *schemaless*, meaning there are no restrictions on fields or types associated with the collection. A file uploaded to this database can be subdivided into smaller pieces known as chunks. Retrieving these chunks individually can allow for efficient arbitrary decryption of very large files without having to load much of the file into memory at once.

[3]JavaScript Object Notation.

The chunks related to a file are uniformly sized except for the last chunk, which is only as large as necessary to contain the data. Each file will also have a small index file that keeps track of the number of chunks, their size, and the name of the original file. The chunks themselves will have a sequence number and the ID of the index file. The structure of the index file can be seen in Figure 2.3, while the structure of the chunks themselves is presented in Figure 2.4.

Placing indexes on the relevant attributes, like *index_id* and *n*, can speed up the time it takes to fetch the correct documents.

```
{
    "id" : UID,
    "file_name": string,
    "chunk_size": number, // bytes of data per chunk
    "length": number, // number of chunks in file
    "upload_date" : datetime,
    "metadata" : any,
}
```

Figure 2.3: Structure of index file

```
{
    "id": UID,
    "index_id": UID // foreign key to index file id
    "n" : number, // sequence number, starting at 0
    "data" : binary,
}
```

Figure 2.4: Structure of chunk

2.3.5 External solutions

A common way of dealing with both media files and applications themselves is to host them externally. Cloud storage services like Google Cloud [34] and content delivery networks (CDNs) like Amazon CloudFront [35] have become popular and can provide many benefits to clients hosting a service. There are usually guarantees on uptime and storage redundancy, as well as geographically spaced servers to reduce latency, without compromising the ease of configuration. Providers like these offer cloud-based

file storage, RDBMS, and NoSQL services that can be configured according to the client's wishes.

Although solutions like these are popular, their underlying implementations are often proprietary. One can not necessarily know what file systems and storage solutions are used[4] in the underlying system. It is also worth noting that external solutions are not always applicable for systems storing sensitive information [36], depending on the solution itself and the geographical origin of the system.

[4]Some cloud solutions offer configurable virtual machines where the client can choose operating system etc.

Chapter 3

Methodology

This chapter explores the different methods chosen to access encrypted data. While all three methods will be used with AES in CBC mode, they have meaningful differences, which will be shown in the benchmarking.

The three methods we want to discuss are as follows:

- Reading and decrypting entire files from disk (NTFS)
- Reading chunks from the disk and decrypting them separately
- Using a document-oriented database with a chunk-based approach to file storage

3.1 Defining solutions

We aim to compare the baseline encryption and decryption methods with a document-oriented approach. Thus, we find it essential to define them properly.

3.1.1 *Disk* solution

The Disk method involves reading entire files from the disk and encrypting/decrypting them in one go and is considered the baseline approach. This is quick and efficient for smaller files, and very simple to implement. It is expected that the execution time of this approach should scale linearly with the size of the input files — unless the system needs to use secondary memory (i.e., not RAM) during execution. One can therefore assume that for *very* large files, delays will be incurred due to higher data fetch latency.

The size of an encrypted file will be almost identical to its unencrypted counterpart, except for the additional initialization vector of 16 bytes and the padding schema employed by the AES cipher, ranging between 1 and 16 bytes. Encrypted file size using this method is seen in Equation 3.1, where IV denotes the initialization vector for the first block (16 bytes), and $Padding$ fills the remainder of the last block as seen in Equation 3.2.

$$FileSize_{encrypted} = IV + FileSize_{unencrypted} + Padding \tag{3.1}$$

$$Padding = 16 - (FileSize_{unencrypted} \bmod 16) \tag{3.2}$$

3.1.2 *Chunk* solution

In this solution, the files are stored in a slightly different manner. As explained in Subsection 2.1.3, the AES encryption algorithm CBC mode encrypts files by dividing them into 16-byte ciphertext blocks. Due to the nature of cipher-block chaining, each of these blocks is reliant on the ciphertext from the previous block. The first block is an exception, as it uses an initialization vector instead.

During encryption, the files are divided into chunks of specific sizes. Assume in this example that we are using 256KB chunks. The initial chunk will have an initialization vector of 16 bytes, as well as 256KB−16B of encrypted data. The next chunk of the file (256KB−16B) will be encrypted using the last 16 bytes of the first chunk as the initialization vector. Those 16 bytes will be prepended to the current chunk (making it 256KB), and the process is repeated until the whole file is encrypted. The chunks are still stored as a single file on the file system but are interpreted differently by the software running the tests. Files encrypted in this way will be slightly larger, as each chunk stores 16 duplicate bytes. The encrypted file size is given by Equation 3.3, where 16 signifies the block size of the AES cipher, and *Padding* is described in

Equation 3.2.

$$FileSize_{encrypted} = IV + FileSize_{unencrypted} + \lceil \frac{FileSize_{unencrypted}}{ChunkSize} \rceil * 16 + Padding \tag{3.3}$$

Since each chunk is stored alongside its IV, it is not required to fetch the preceding chunk to decrypt a random chunk. Therefore, an assumption is made that this solution will have improved performance in terms of time to first byte (TTFB), and hopefully hardware utilization since there is no need to store the entire file in memory.

3.1.3 *DBC* solution

This solution is comparable to Chunk, except the files will be stored and retrieved through a database. Due to this, we have named the solution Database Chunk, or DBC for short.

After a file has been encrypted it is uploaded to a document-oriented database which handles files by subdividing them into chunks. To fit the previous solution, the database will be configured to store files in chunks of the same sizes as Chunk during testing.

For each chunk to be fully decryptable without reading the prior chunk, this solution requires the encryption method from the Chunk solution. Therefore, this solution does not have its own encryption method and will instead inherit it from Chunk.

The system running the tests will host both the code and the database, meaning network latency will have the least possible impact. It may be expected that this solution takes somewhat longer to fetch data due to the software overhead. The first read of a session might also take longer than average due to the database being "cold", i.e., not having any data in the cache.

3.2 Performance indicators

Given that our solutions are meant to compare the aforementioned strategies of accessing encrypted files, we need a list of established metrics (performance indicators). These lay the foundation for the tests and show how well the solutions perform against each other. The performance indicators are listed here in no particular order.

3.2.1 Time to first byte

Time to first byte (TTFB) is a metric most associated with web development and signifies the time it takes to receive the first byte of a resource after requesting it from a server [37]. Several factors can cause the TTFB to vary significantly. These include the physical distance between the client (the requestor) and the server, server load at a particular time, signal hindrances (e.g., bad weather), and the ISP[1] being overloaded. Since we are testing only what happens on the server, we are unaffected by these factors.

In the case of this project, TTFB will be a valuable performance indicator. This indicator will essentially give a measure of how much time it takes between a request being sent and a decrypted response being ready.

3.2.2 Throughput

Throughput measures the amount of data being transferred over a given time span and is a good measure of the performance of a given system. Each solution's throughput is measured and compared for readable relative comparisons. For media streaming purposes, the throughput of a given decryption algorithm should be higher than the bitrate to avoid users having the decryption process bottlenecking the service.

Throughput is also related to the next performance indicators: lower throughput may adversely affect system resources.

[1]Internet service provider, often a telecommunications company, that provides internet access to clients.

3.2.3 Memory usage

All servers are limited by their physical resources, which include their main memory. This is a physical component called random access memory (RAM). Being one of the fastest memory components on the computer, it is vital that memory consumption stays below that of the RAM capacity. If an application requires more memory than what is available on the system, the operating system may use paging in order to allow the application to use more storage [38]. Paging refers to the use of secondary memory (e.g., the HDD or the SSD) as if it were main memory. The secondary memory will, in almost all cases, have lower bandwidth, suffer from worse read and write speeds, and be placed physically further away than the main memory, slowing down the system altogether.

While an algorithm could achieve a speed-up over another, it might be sacrificing memory space in a *space-time tradeoff*. This tradeoff could cause problems on systems with limited RAM, making it important to monitor the memory utilization of our solutions.

The benefits of having a lower memory footprint during decryption can therefore be:

- **System performance:** higher decryption speed due to "correct" memory utilization
- **System stability:** background tasks and processes are less affected
- **Scalability:** if the memory usage is low, it facilitates a more scalable system, and perhaps more concurrent decryption processes at once

3.2.4 Other performance indicators

Several other performance indicators could be valuable to analyze in the context of encrypted video streaming. Some of them are named here, but they will not be discussed in the analysis.

CPU usage

The central processing unit (CPU) is the physical component that takes care of most, if not all, computations on a system. The process of encryption and decryption are often CPU heavy and require it to do altercations to the data. In the case of AES encryption, these are primarily bitwise operations. A bitwise operation is normally easy for a CPU to handle and can be as simple as a negation (e.g., the binary value $0101 \rightarrow 1010$). However, due to the amount of these operations needed for larger files, the CPU could become a bottleneck.

Parallelism

In multicore systems, it is possible to utilize the processing cores in parallel, speeding up tasks by dividing the labor. The performance gained depends on the implementation and the degree of parallelization of the program [39]. This is an important indicator since most modern processors have two or more processing cores. However, we decided to omit it from the project due to the increased complexity of the test framework, overhead associated with thread initialization, and not being as important as the previously mentioned performance indicators.

Storage medium degradation

When files on a system are created, altered, and deleted, the underlying file system must decide where the new data should be placed. When hard disk drives (HDDs) are used, these files typically align sequentially, so the seek times on the disk platter are kept to a minimum. However, when a file is appended with new data after another file is already stored after it (physically, not necessarily temporally), the appendix may be placed after the last file. This phenomenon is called fragmentation [40].

When the drive is fragmented, HDDs, in particular, can suffer from worse read performance. This is because of the physical limitations of the moving parts that must traverse the spinning platters [41]. Since the contents of a file are not necessarily stored sequentially on the disk, the hard disk head takes more time to read out the whole file. Fragmentation can occur in SSDs as well, but they are not as affected due to them not having moving parts and faster read times in general.

A note on solid state drives (SSDs) is that they can suffer from *write amplification* (WA). WA is an undesirable side effect because SSDs must rewrite entire blocks on deletes [42]. This can happen when updating or writing to the drive (and when garbage collection occurs) and requires the entire block to be read and updated before being written to another block.

An example is when an SSD chooses to write new data to a block that already has data written to it. The previous data of the block must then be rewritten into a new block before the new data is written to the "current" block. The user only intended one write operation, but two writes were needed to accomplish the single write. Although most SSDs use wear leveling to increase the life span of the drive, write amplification can cause the drive to be shorter-lived.

Even though these hardware-specific degradation metrics are measurable, it is hard to give a fair evaluation without more knowledge on the topics of operating systems and drive firmware interoperability. It does also stray out of the scope of the research question.

3.3 Test bed

The test bed is an application written in Python. It is designed to analyze the different solutions, focusing on the aforementioned performance indicators. The application will also need to be profiled, a process from which we can extract information on parts that need to be optimized. By doing this, we can remove rudimentary function calls that are not part of the actual benchmarking.

3.3.1 Choice of programming language

Several programming languages were considered for creating the test bed. In order to make a suitable choice, we looked at ways the different languages could help with giving good results without compromising efficiency. The language would also need support for relevant libraries[2] (e.g., the AES-CBC algorithm).

[2]Pieces of external code that can be implemented into a codebase. Examples include algorithms in cryptography, mathematics, and tools for visualization.

We first looked at Rust, a relatively new programming language praised for its runtime performance and memory efficiency [43]. Having been one of the most favored programming languages by developers, according to a yearly survey by StackOverflow [44], it has generated a community creating libraries that support most use cases. Rust had everything needed in order to create a test bed, and seemed like a great choice. There was, however, one caveat: lack of experience using the language. This created a barrier since the scope of the application was somewhat large, and the time given to develop it was limited.

We also looked into C++. Although we were more experienced with C++ than Rust, choosing it could compromise time spent on data analysis by spending too much developing and squashing bugs along the way.

Python was the programming language we had the most experience with, having used it in several projects earlier. The package manager had support for all relevant functionality. Contrary to Rust and C++, Python seemed like the language that would give the fastest progress in terms of testing and benchmarking.

There were concerns as to the runtime performance of Python as opposed to the other languages. Programs that rely heavily on arithmetic operations may suffer due to the Python interpreter's way of dealing with the boxing of operands and containers [45]. This is relevant since the AES cipher uses block ciphers in order to encrypt and decrypt data, leveraging many bitwise operations [46].

To leverage the speed of implementation using Python, and the performance benefits of C++, we found Python libraries that utilize C[3] for functions that are critical for performance (see Subsection 3.3.2). Hence, Python was the language of choice for the test bed.

3.3.2 Software breakdown

To make the benchmarking reproducible, we noted down different software versions used in the test bed. This includes Python itself, and libraries that were used to speed

[3]The language C++ is based on. Although more primitive, it is known for being highly performant.

up development and ensure correct implementations. Vetting of the implemented libraries was also an important point since the usage of external pieces of code can introduce bugs and security issues. Table 3.1 shows a compacted view of the software used in the program. *slib* denotes the standard library, meaning the version is the same as the Python version.

The *time* module[4] was used in order to achieve high-resolution timing inside function calls [47]. This was needed both to gather results when timing different operations and to extrapolate the time needed to perform larger test batches. The *perf_counter()* function was used to measure the time differences.

$cProfile$ and *pstats* both deal with the profiling of the application [48]. The $cProfile$ code is wrapped around parts of the existing code base, and executed simultaneously as the benchmarking runs. While running, it gathers statistics and writes them to a file. *pstats* is used to extract and analyze the data provided by $cProfile$, and is used in scripts running after the benchmark is complete. As mentioned, analyzing the statistics from these packages helps find function calls that should be disregarded in the results. Examples include printing to the standard output, the function call to the $cProfile$ executor itself, and arbitrary functions deemed unnecessary for the application.

We chose to use *PyCryptodome* for functions related to the AES cipher [49]. It is a fork[5] of a no longer maintained package called *PyCrypto*. The package contains many encryption methods, and leverages code written in C for performance-critical operations as mentioned in Subsection 3.3.1. There are several reasons why the C code runs faster, some of them being:

- The C code is compiled, whereas Python code is interpreted. Compilation is the process of turning the code into machine code through a compiler, leaving an executable. Compiled programs tend to run faster than interpreted programs [50].

[4]Essentially the same as a library. Module, package, and library are used interchangeably in this thesis.

[5]In relation to software version control, a fork is a copy of an existing repository.

Software	Version	When used
Python	3.10.6	During execution
time	*slib*	During execution
cProfile	*slib*	During execution
pstats	*slib*	Post-processing
PyCryptodome	3.17	During execution
unittest	*slib*	Prior to execution
tracemalloc	*slib*	During execution

Table 3.1: Software and libraries used.

- C does not have a garbage collector (GC) contrary to Python. Garbage collection is a process of proactively deallocating memory during runtime. It can eat resources, and it's hard for developers to evaluate performance degradation due to the GC.
- Python dynamically allocates memory on the heap, a process that tends to be less efficient than using the stack. C code more often makes use of the stack, a more efficient, yet smaller, memory component.

The *unittest* library was used in order to simplify making unit tests, a topic that will be discussed later [51].

Lastly, *tracemalloc* was used to gather statistics on memory usage during runtime [52]. After importing the library, we start tracing the relevant pieces of code and log it after it has finished. The most pertinent information *tracemalloc* gives is the current memory allocated and the peak memory allocated during execution.

3.3.3 Hardware breakdown

Testing was further standardized by using the same hardware for all benchmarking. The tests were run on a computer with the specifications listed in Table 3.2.

The machine was running on Windows 11 Home, version 22H1. The file system associated with the operating system was NTFS.

[6] 4 performance cores, 8 efficiency cores.

Component	Specifications
CPU	12 cores[6], 4.5GHz max frequency
Main memory (RAM)	LPDDR5, 16GB, 4800MHz frequency
Secondary memory (SSD)	M.2, 512GB storage

Table 3.2: Hardware components of test bench.

3.3.4 Test parameters

For testing, we are using different file and chunk sizes to identify differences properly. This can also help find break-even points for the solutions as they might have strengths and weaknesses depending on the sizes.

Different file sizes

According to a technical report by Microsoft [53], performance differences exist when storing files on NTFS directly versus as BLOBs on SQL Server. In the report, they mention that for files smaller than 256KB, SQL Server handles them more efficiently as BLOBs. For files larger than 1MB, storing them directly on NTFS is more efficient. When discussing efficiency in the report, they focus on read/write throughput and disk fragmentation — HDDs were used for testing. They also mention that the break-even point will vary depending on several factors.

For the purposes of this study, the file sizes mentioned in the report are considered too small. Since the main goal is discovering efficient solutions to read and decrypt large media files, we set the lower limit to 16MB. From this, the sizes increased exponentially to 32MB, 64MB, 128MB, etc. The upper limit was set to 4GB.

Varying chunk sizes

For the *Chunk* and *DBC* solutions, we have chosen to try two different chunk sizes. We have chosen to go for 256KB and 4MB. These will both be tested for all the different file sizes.

The reason for testing different chunk sizes is that it may affect runtime performance and resource usage. In terms of runtime performance, certain chunk sizes may out-

perform others by utilizing the underlying file system more efficiently. Smaller chunks will likely lead to less memory expenditure but require more reads from the disk or database. It could also help close in on a potential break-even point for the most efficient chunk size per file.

3.4 Benchmarking solutions

To assess the different solutions, we will run extensive tests. These will produce valuable insights regarding the performance indicators we have chosen to focus on; see Section 3.2. The test bed, as mentioned, is written in Python and uses the packages mentioned for profiling, timing, and measuring resource usage.

3.4.1 Ensuring validity of the tests

Since the thesis largely depends on the results from the benchmarking, it is important to ensure that the results can be trusted. There are several measures to look at when trying to provide this validity. Examples include ensuring that functions in the application return the correct results and that the machine running the tests is not running heavy background tasks while performing benchmarks. Here we will present measures taken to have a reliable test environment.

Data integrity and determinism

Tests were written to ensure data integrity after encrypting and decrypting data. Since AES-CBC makes use of an initialization vector (see Subsection 2.1.3), a pseudorandom value generated during runtime, encrypting the same data will twice result in different ciphertexts. This is known as a non-deterministic function, where the same input does not necessarily give the same output. However, upon decryption, the plaintext will still be the same. The data integrity tests were implemented to validate that no two ciphertexts were identical and that the plaintext provided by the decryption functions matched the original plaintext exactly.

Manual tests to ensure data integrity were also employed. This was done by using media files — sound, image, and video — checking for corruption and visually assessing

the decrypted ciphertext.

Profiling the test bed

As was noted in Subsection 3.3.2, the *cProfile* and *pstats* libraries were to be used in the benchmarking. In addition to this, they were used to identify pieces of code that were especially time-consuming. Using this data, we were able to improve certain parts of our code base.

It should be noted that running a program with a software profiler attached negatively affects performance. This is because the profiling software (at least *cProfile*) works as a wrapper around the existing software. It then has to probe and log information regarding the software being executed, increasing the computational effort. We found that *cProfile* did not have a too big footprint, contrary to the *profile* library, which is implemented solely in Python.

Unit tests

Unit tests are essentially a way of testing a "unit of code". In most cases, this includes methods and functions but differs from larger-scale system tests where the entire life cycle is tested. These tests were implemented by asserting that the output of functions with specific inputs was of certain values. For encryption-related functions, predetermined initialization vectors were used so the non-deterministic nature would not interfere.

Resource surveillance

Another important point for getting accurate test results was to monitor the machine running the benchmarks. Operating systems like Windows 11 have automated ways of performing background tasks and updates that can affect performance. Although these tasks are not necessarily resource intensive, it is worth aiming to have as little interference as possible during testing. Therefore, measures were taken to mitigate the impact: all non-critical system tasks were ended, automatic updates were deactivated, and internet access was turned off.

During testing, only *PowerShell* (version 5.1) was open and used to run the tests, except *Task Manager*, which was used during some runs to validate resource usage in accordance with benchmarking results. PowerShell is a pre-installed command-line shell that comes with most new Windows installations, from which scripts and programs can be run [54]. Task Manager also comes built-in with Windows and provides statistics associated with hardware, like the CPU.

Repeated tests

Even with the aforementioned measures in place, running a single test per solution is not enough. Coincidences happen, and *one* good result may not resemble the actual performance. We chose to do at least 300 run-throughs for each combination (e.g., *DBC* with 64MB file and 256KB chunks). With this, it's easier to see trends and accurately depict their respective performance.

3.4.2 Benchmarking application

The benchmarking application is meant to give accurate measurements of runtime performance and resource expenditure. The main components of the application and how they are connected are detailed in Figure 3.1. Although there are more files associated with the application, the ones of interest are:

- Encryption and decryption: *aes_cipher.py*
- I/O handler: *binary_file_handler.py* (and *file_handler.py*)
- Configuration: *config.py*
- Executables: *benchmark.py*, *profiler.py*, and *resource_profiler.py*
- Document-oriented database driver

Encryption and decryption

The file *aes_cipher.py* contains a class called AESCipher. Its job is to encrypt and decrypt binary data. Although one can encode data with Base64 to make the output more legible, it was deemed unnecessary. This is because the process in itself takes

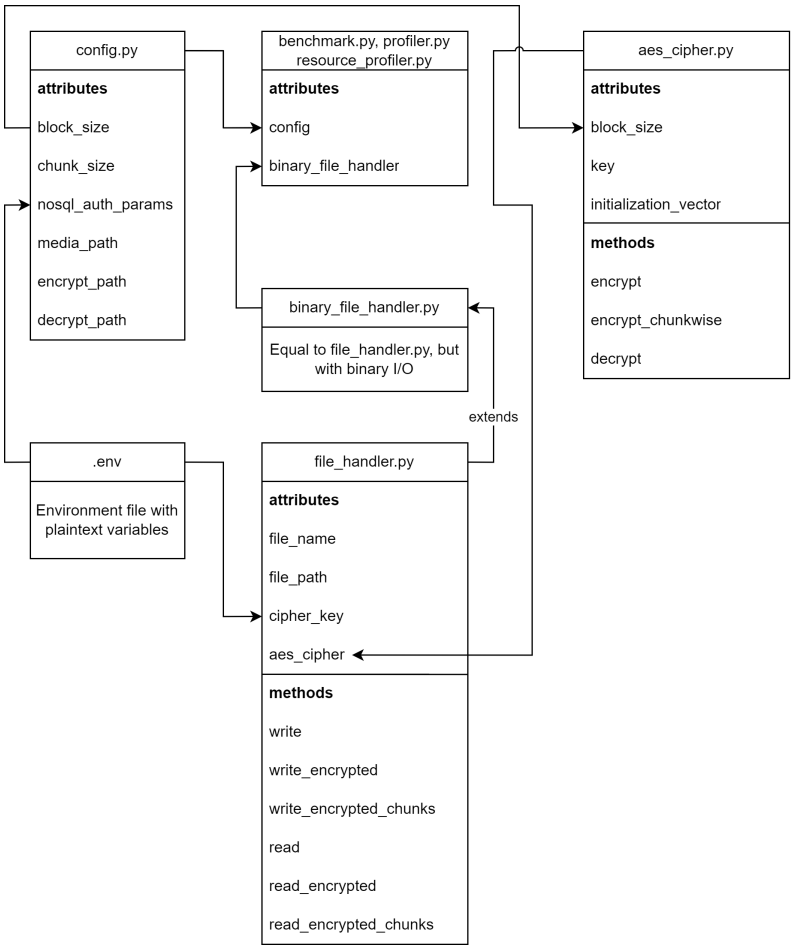


Figure 3.1: Relationship diagram of the application, highlighting the most important aspects.

time, and files become larger (an increase of at least 33% [55]) and therefore require more processing to be done.

AESCipher is instantiated with a key, namely the symmetric key used for encryption and decryption. The block size is given by the algorithm — 16 bytes — and the initialization vector is created pseudo-randomly when using encryption methods and inferred when using decryption methods.

Its methods for regular encryption and decryption (*encrypt* and *decrypt*) use a standard implementation provided by the *pycryptodome* library. The encryption method is provided with a raw byte string (plaintext), which is promptly padded to be a multiple of the block size (16). An IV is generated and used together with the key and mode of operation to create an instance of *pycryptodome* AES. It is then used to encrypt[7] the raw data into ciphertext before prepending the plaintext IV and returning the result.

The decryption process is similar. It is also fed with a byte string (ciphertext) as the parameter. First, the first 16 bytes are extracted, which is the IV for the ciphertext. The *pycryptodome* AES is then instantiated in the same manner as encryption. The rest of the byte string (excluding the IV) is then decrypted[8], before un-padding the resulting plaintext.

Our implementation of chunk-wise cryptography is solved similarly and is touched on in Subsection 3.1.2. Figure 3.2 shows the iterative nature of the chunk-wise encryption method. The overhead in terms of memory usage as seen in the figure is explained by Equation 3.3, and is stored in the last chunk. The decryption method can then be used on individual chunks, as their respective IV is stored in the chunk.

I/O Handler

The files $binary\_file\_handler.py$ and $file\_handler.py$ each contain methods for reading and writing data to files; in binary mode and text mode respectively. The binary mode was preferred for benchmarking (as explained in Subsection 3.4.2: Encryption

[7]Using the *pycryptodome encrypt* method.
[8]Using the *pycryptodome decrypt* method.

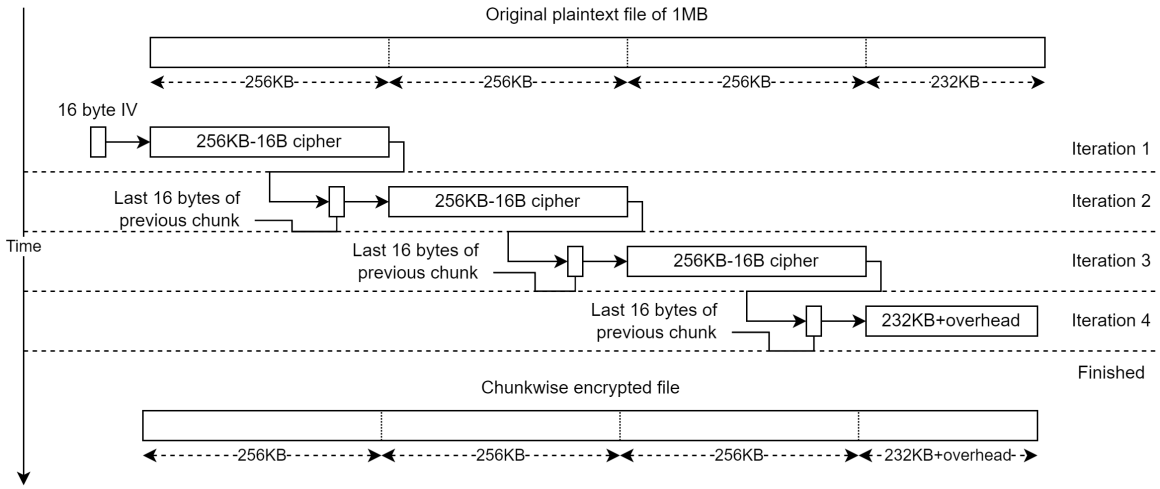


Figure 3.2: Flow of the chunk-wise encryption algorithm.

and decryption), while the text mode was used during testing.

A class named BinaryFileHandler exists in *binary_file_handler.py*, which contains all methods related to I/O. It is instantiated with a filename and associates this filename with relevant paths, such as the path it should be stored at when after encryption. The BinaryFileHandler creates an instance of the AESCipher for all methods requiring either encryption or decryption and uses AESCiphers methods according to the specified solution (whole file or chunk-wise).

The built-in *open* function in Python is used, alongside context managers to close files when reads/writes are finished and to deallocate memory.

Configuration

config.py contains mostly all configuration variables and is read by the executables at runtime. It specifies important values, namely what chunk size should be used, encryption keys for the files, and several paths. Although these variables could be hard-coded in different places, it could have led to bugs and unnecessary time waste.

Executables

There are three main entry points in the application, each providing statistics on different performance indicators. They are the following: *benchmark.py*, *profiler.py*, and *resource_profiler.py*. Each of them is run through the terminal with specific command line arguments, such as file name[9] and the number of iterations. An example of how *benchmark.py* could be run 100 times on a 1GB file can be seen below. After running this command, the process asks which solution should be benchmarked and runs the chosen solution.

```
> py .\benchmark.py 1GB 100
```

For timing the different solutions, *benchmark.py* was used. It only relies on the time library and logs the relevant timings to a file. *profiler.py* is almost identical to *benchmark.py*, but the timings are removed and it uses the *cProfile run* function to wrap the executing functions for profiling. *pstats* was then used to collect the data provided by the profiling.

To accurately get information regarding resource usage, we also created the file *resource_profiler.py*. Although similar to the previous two, it was made to run with *tracemalloc*.

Document-oriented database driver

Using a popular document-oriented database, we implemented a solution for storing binary file data and metadata, as mentioned in Subsection 2.3.4. The driver needed to support file uploads and partial file downloads.

Uploads were made possible using a library provided by the document-oriented database creators. Partial file downloads used seeking, which also came with the library.

[9]For the benchmarking, our file names coincide with their size.

Chapter 4

Results and Analysis

The data presented in this chapter are the results of the benchmarking process outlined in Section 3.4. Each solution was tested sequentially on the same system, ensuring an environment with an adequate level of consistency. We ran our tests with a range of differently sized inputs as described in Subsection 3.3.4. As described in Section 3.1, we will refer to the solutions by their shortened aliases: Disk, Chunk, and DBC. We will specify chunk sizes used with Chunk and DBC in the format *name-chunksize*.

Results will be split by category, with encryption first and decryption second. The analysis will be in the last section.

4.1 Encryption

As mentioned in Subsection 3.1.3, DBC is not eligible for encryption, instead inheriting its results from Chunk. The average amount of time it took to encrypt the differently sized files can be seen in Table 4.1 and Figure 4.1, while the throughput measured in megabytes per second is presented in Table 4.2 and Figure 4.2. Encryption memory utilization is shown in Table 4.3 and Figure 4.3.

Filesize	Disk	Chunk-256KB	Chunk-4MB
16MB	0.04531	0.05776	0.04695
32MB	0.11183	0.12547	0.10053
64MB	0.25408	0.22865	0.1992
128MB	0.4528	0.48101	0.39269
256MB	0.9463	0.69397	0.78947
512MB	1.98176	1.7599	1.60428
1GB	3.93614	3.80621	3.42833
2GB	8.24366	7.81832	6.7826
4GB	49.98934	16.8439	17.08196

Table 4.1: Encryption time in seconds

Filesize	Disk	Chunk-256KB	Chunk-4MB
16MB	353.122931	277.00831	340.78807
32MB	286.148618	255.04105	318.31294
64MB	251.889169	279.90378	321.28514
128MB	282.685512	266.10673	325.95686
256MB	270.527317	368.89203	324.26818
512MB	258.356209	290.92562	319.14628
1GB	260.153348	269.03403	298.6877
2GB	248.433342	261.94886	301.94911
4GB	81.937469	243.17409	239.78513

Table 4.2: MB/s encrypted

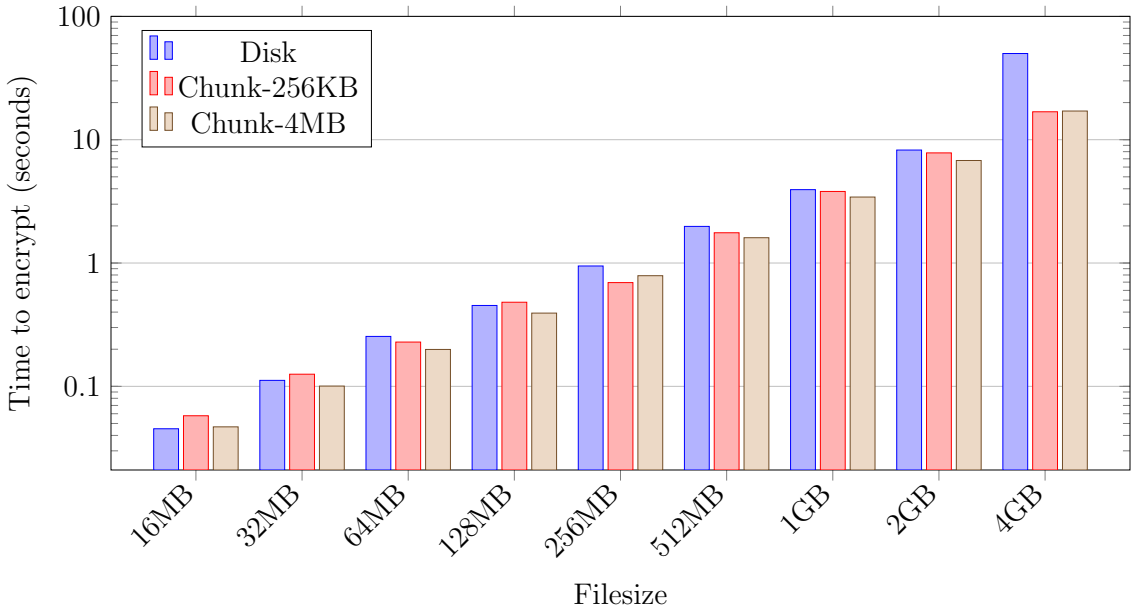


Figure 4.1: Average encryption time (logarithmic scale)

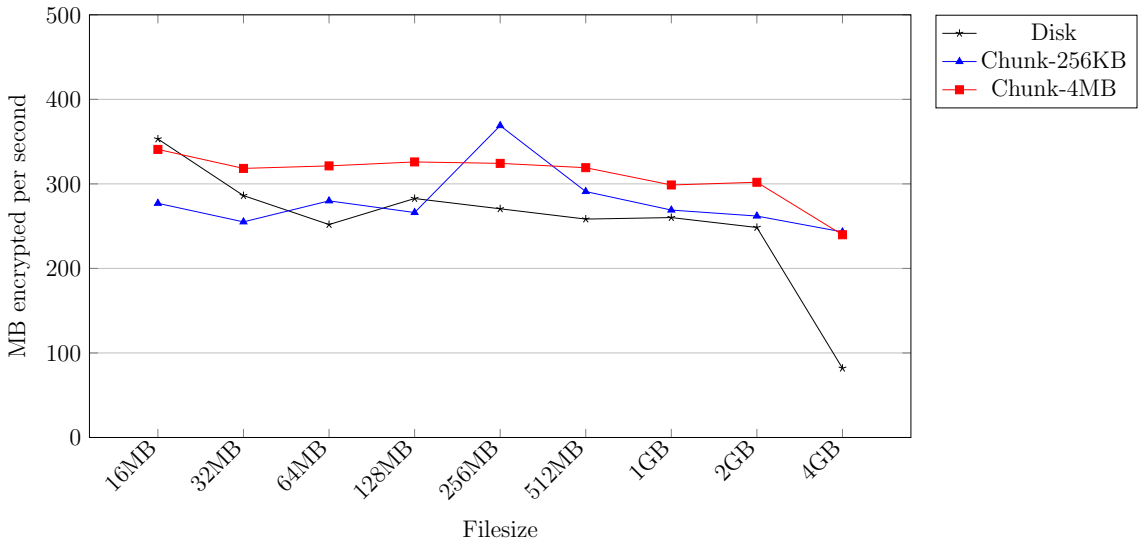


Figure 4.2: Average encryption throughput

Filesize	Disk	Chunk-256KB	Chunk-4MB
16MB	64.017MB	1.0096MB	15.957MB
32MB	128.017MB	1.0096MB	15.958MB
64MB	256.017MB	1.0104MB	15.959MB
128MB	512.017MB	1.0105MB	15.959MB
256MB	1024.017MB	1.0105MB	15.959MB
512MB	2.0GB	1.0105MB	15.959MB
1GB	4.0GB	1.0107MB	15.960MB
2GB	8.0GB	1.0108MB	15.960MB
4GB	16.0GB	1.0108MB	15.960MB

Table 4.3: Encryption memory peak

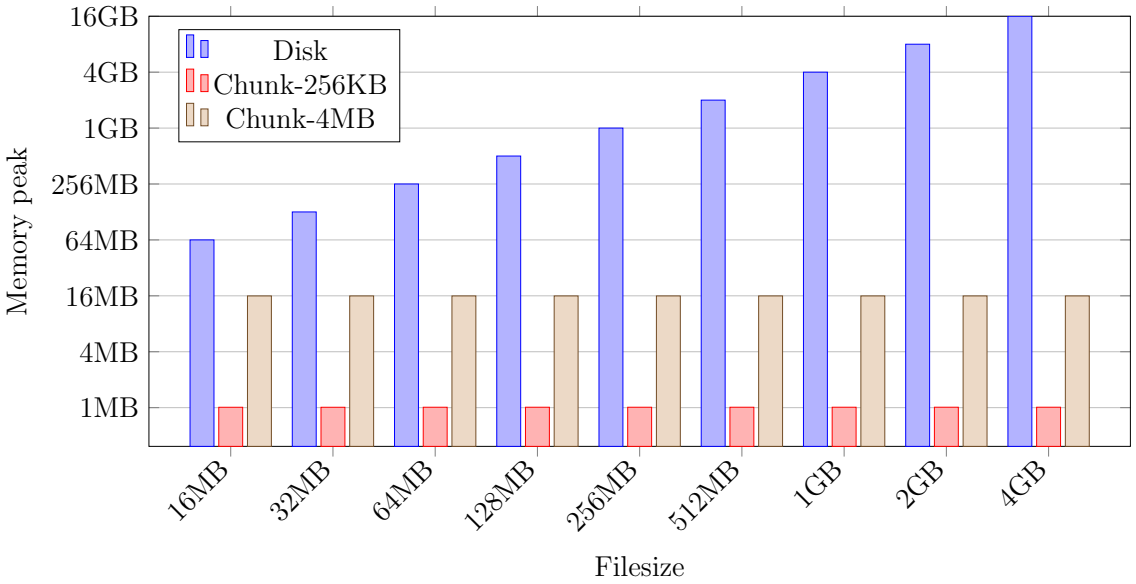


Figure 4.3: Memory peak during encryption (logarithmic scale)

4.2 Decryption

Average decryption time is shown in Table 4.4 and Figure 4.4. Average decryption throughput can be seen in Table 4.5 and Figure 4.5. Decryption memory utilization is shown in Table 4.6 and Figure 4.6.

Filesize	Disk	Chunk-256KB	Chunk-4MB	DBC-256KB	DBC-4MB
16MB	0.04454	0.11638	0.04202	0.13299	0.07398
32MB	0.12013	0.1213	0.08846	0.13786	0.13223
64MB	0.24299	0.22399	0.17624	0.2576	0.26542
128MB	0.46079	0.46687	0.34986	0.48636	0.54158
256MB	0.95609	0.62864	0.71325	0.87164	1.19868
512MB	2.01201	1.71314	1.39712	1.81103	2.19617
1GB	4.13394	3.48021	2.87999	3.51681	4.1094
2GB	8.15399	6.89489	5.87454	7.29861	8.3328
4GB	62.45584	14.78916	11.90777	14.86481	17.16025

Table 4.4: Decryption time in seconds

Filesize	Disk	Chunk-256KB	Chunk-4MB	DBC-256KB	DBC-4MB
16MB	359.228	137.481	380.771	120.310	216.275
32MB	266.378	263.809	361.745	232.120	242.003
64MB	263.385	285.727	363.141	248.447	241.127
128MB	277.784	274.166	365.861	263.180	236.346
256MB	267.757	407.228	358.920	293.699	213.568
512MB	254.47	298.866	366.468	282.712	233.133
1GB	247.706	294.235	355.557	291.173	249.185
2GB	251.165	297.032	348.623	280.601	245.776
4GB	65.582	276.960	343.977	275.550	238.691

Table 4.5: MB/s decrypted

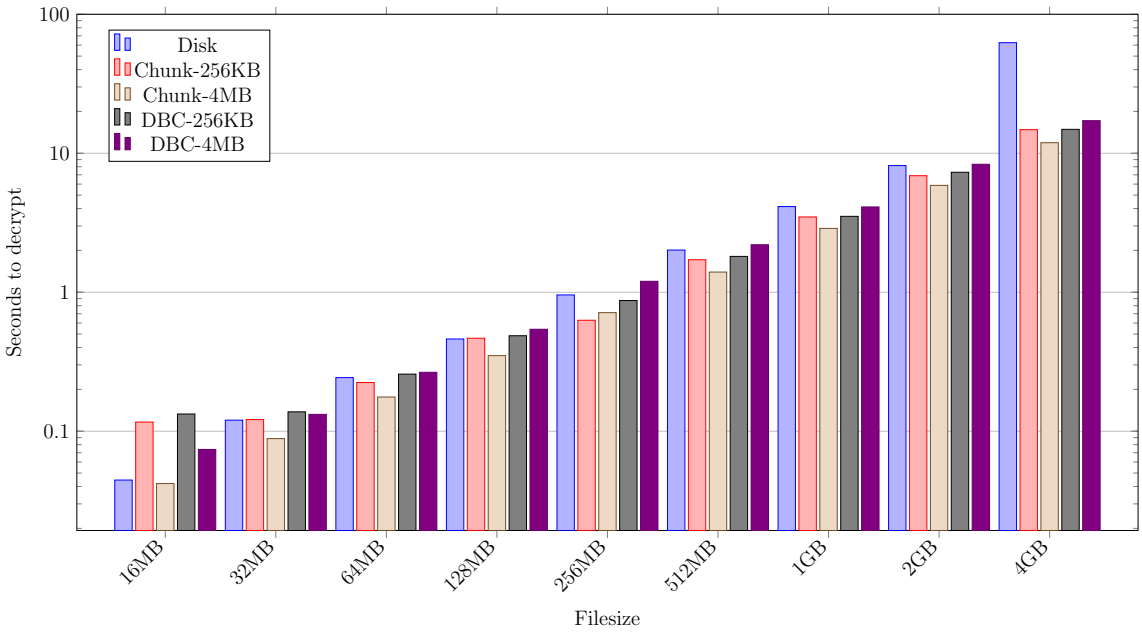


Figure 4.4: Average decryption time (logarithmic scale)

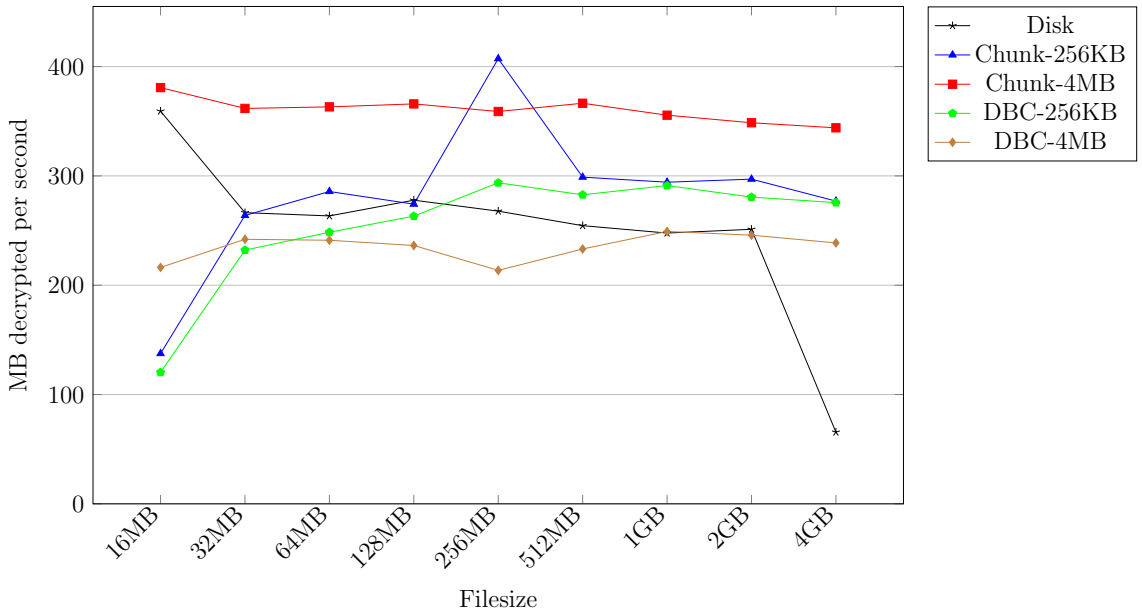


Figure 4.5: Average decryption throughput

Filesize	Disk	Chunk-256KB	Chunk-4MB	DBC-256KB	DBC-4MB
16MB	48.0186MB	0.766MB	11.972MB	32.076MB	32.049MB
32MB	96.0186MB	0.766MB	11.972MB	32.585MB	40.019MB
64MB	192.019MB	0.766MB	11.972MB	32.586MB	40.019MB
128MB	384.019MB	0.766MB	11.972MB	32.592MB	40.020MB
256MB	768.019MB	0.766MB	11.972MB	32.599MB	40.026MB
512MB	1.500GB	0.766MB	11.973MB	32.608MB	40.033MB
1GB	3.000GB	0.766MB	11.973MB	32.626MB	40.047MB
2GB	6.000GB	0.766MB	11.973MB	32.634MB	40.058MB
4GB	12.000GB	0.766MB	11.973MB	32.650MB	40.079MB

Table 4.6: Memory peak during decryption

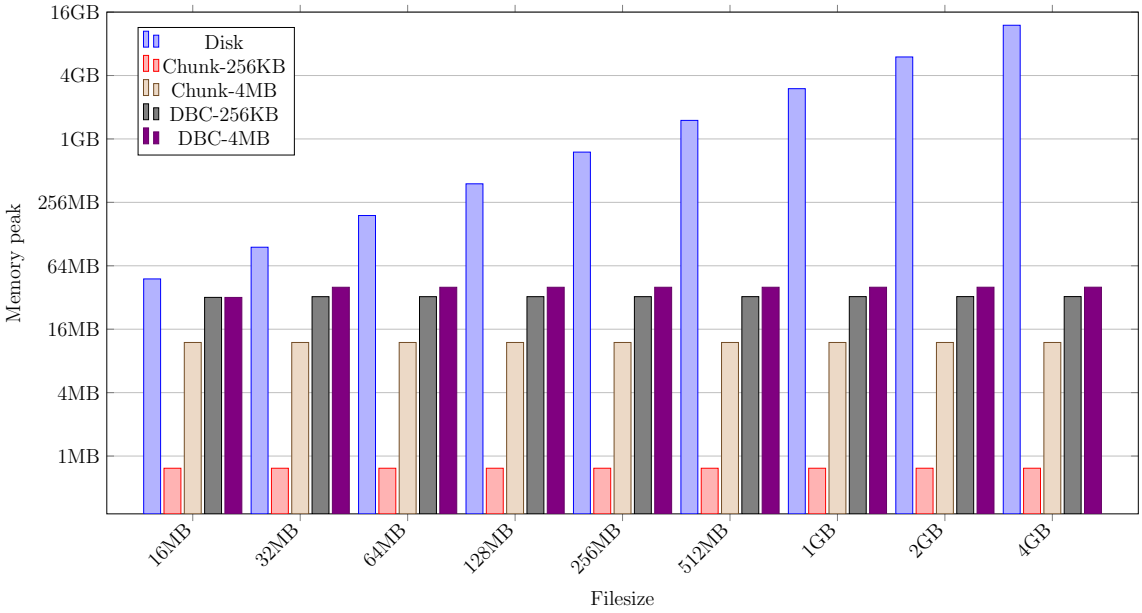


Figure 4.6: Memory peak during decryption (logarithmic scale)

4.2.1 Dry run

A relevant *dry run* pattern was observed solely in DBC. When the database was initialized, the first run of each benchmark was considerably slower than the average. The decryption time of the first run of each benchmark is displayed in Table 4.7 as a percentage of the average decryption time shown in Table 4.4. Both dry run times and average run times were averaged and are displayed in Figure 4.7.

Filesize	DBC-256KB	DBC-4MB
16MB	349%	136%
32MB	235%	213%
64MB	172%	219%
128MB	179%	198%
256MB	122%	303%
512MB	144%	195%
1GB	155%	164%
2GB	167%	231%
4GB	207%	230%

Table 4.7: DBC dry run time compared to average (percentage)

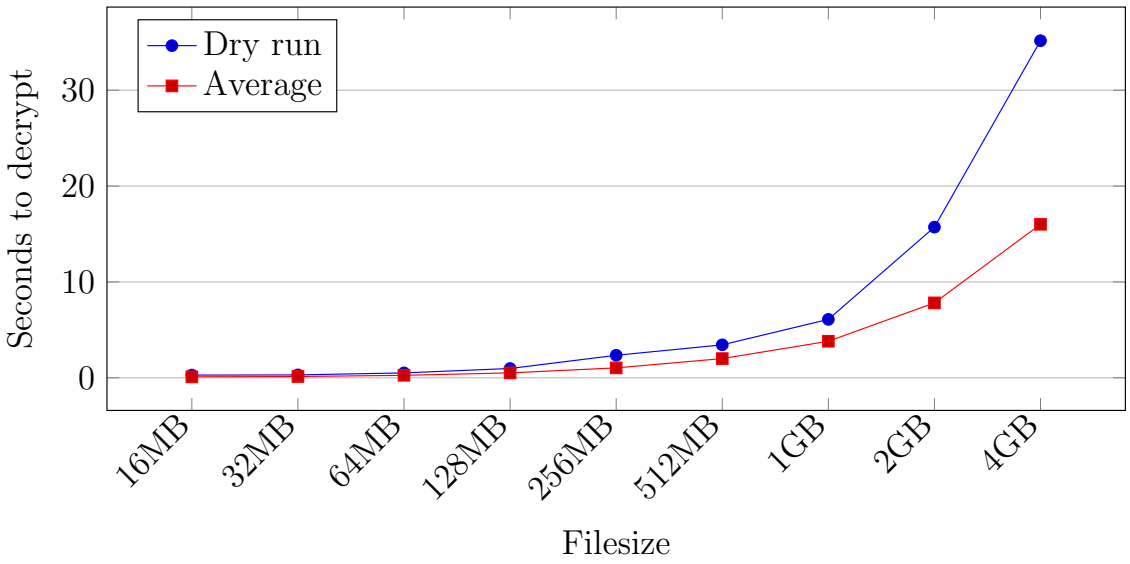


Figure 4.7: DBC dry run compared to average (seconds)

4.2.2 Time to first byte

The chunk-based solutions can deliver any amount of data from any part of a chosen encrypted file on demand. We queried for a single chunk and measured the time between sending the request and receiving the entire decrypted chunk. This data is shown in Table 4.8 and visualized in Figure 4.8.

Method	Milliseconds
Chunk-256KB	0.99
Chunk-4MB	14.08
DBC-256KB	46.33
DBC-4MB	49.11
Dry run DBC-256KB	73.48
Dry run DBC-4MB	75.97

Table 4.8: Time to first byte

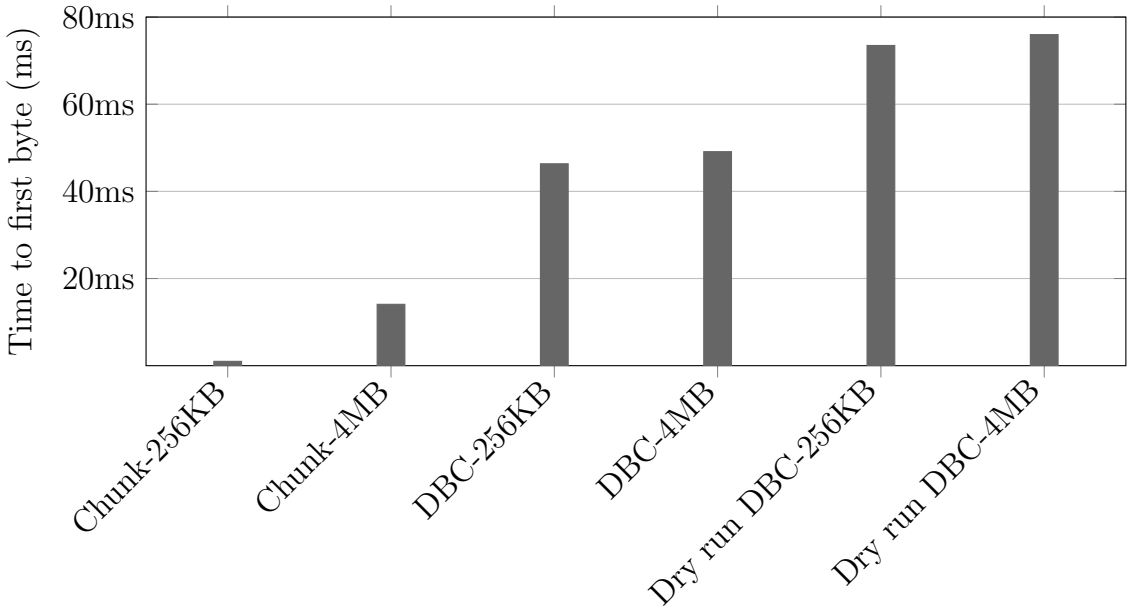


Figure 4.8: Time to first byte

4.3 Analysis

Each solution seems to roughly follow the expected linear trend between file size and decryption time. While Disk performs well at small file sizes, it gets progressively beaten out by Chunk. At the largest file size tested, Disk struggles with performance in a manner that both Chunk and DBC avoid. The exact reason for this is unclear, although there is a likely culprit. Main memory could be overly taxed due to the entirety of both plaintext and ciphertext existing in memory at the same time in the Disk-solution. As Chunk and DBC are more memory-efficient, they do not hold massive amounts of data in main memory at a time. They read a chunk, encrypt/decrypt it, then immediately write the processed chunk to a file before dropping it from memory and moving on to the next chunk.

Brief testing on a different system with 32GB RAM (twice as much as the benchmarking system) saw the linear trend for Disk encryption time upheld at 4GB. The trend was broken once again at 8GB, lending credence to our hypothesis that lack of main memory space is the issue for Disk at higher file sizes. This means Chunk and DBC provide massive performance gains in environments that are starved of main memory when compared to the more naive Disk.

DBC performs consistently worse than Chunk, which is to be expected as they use the same mechanisms for decryption, but DBC has to retrieve chunks from the database instead of finding them directly in the file system. This difference seems to be the main factor causing DBC to spend approximately 25% more time than Chunk in our testing. DBC has the additional drawback of its first run being considerably slower than subsequent runs, likely due to the database being cold. This pattern can clearly be seen in Table 4.7, and was not observed in the other solutions. This was also demonstrated by restarting the database after some runs to see if this pattern persisted, which it did. Although the internals of the proprietary document-database is unknown, it is most likely due to an empty cache.

Chunk seems to outperform Disk for both encryption and decryption quite rapidly as file size increases, even before considering Disk's increasing memory requirement. Before hitting the memory wall, Disk outperforms DBC for decryption at any file

size. DBC weighs up for this by being better suited for progressive download while still performing at an adequate level.

Chunk's TTFB varied wildly with chunk size, with the time for 4MB being roughly 14x the time for 256KB, which roughly matches their size difference. For DBC, there was an increase in TTFB when chunk size increased, but it was much less significant. This means that if the ideal chunk size for decryption is on the larger side, TTFB will in all likelihood not suffer much.

As mentioned in Subsection 2.2.2, the roof for common bitrates is 300Mbps. As bitrate is specified in decimal, the corresponding binary value is 35.76MB/s. As DBC's throughput generally ranges from 200MB/s to 300MB/s, it is safe to say that the decryption process should not be a bottleneck for the delivery of most media through a document database. This means that DBC is viable for use in VOD-systems working on encrypted files up to a very high bitrate, as a decryption throughput of 200MB/s translates to a bitrate of 1677Mbps.

Chapter 5

Conclusion

This thesis has compared different ways of storing and serving encrypted files, focusing on the utilization of chunks. It used a simple approach that read files from disk in one uninterrupted operation (Disk), an approach allowing random access decryption from disk using chunks (Chunk), and a variant of the chunk method using a document database (DBC). We hypothesized that these chunk-based methods would have a comparable decryption throughput to Disk, while also having additional benefits such as progressive decryption and random access decryption.

We have shown that for files that range from 16MB to 4GB, the chunk-based decryption methods have a comparable and at times superior throughput compared to Disk. While Disk runs into issues when encrypting file sizes of approximately 1/4 of main memory capacity, the memory requirements of both Chunk and DBC hardly scale with file size at all. While Chunk outperforms DBC, the difference is not very pronounced. As many systems require a database solution, DBC can be a viable option

Storing encrypted media files in chunks in a document database and decrypting on the fly is a promising approach, allowing systems to combine at-rest-encryption with services such as video on demand without the heavy latency incurred by using a simpler method.

5.1 Future work

The research provided in this thesis gives indications as to how the different solutions perform. Still, there are points of research that could be investigated in order to get a better overview.

5.1.1 Measurement of CPU times

While the Chunk method seems to outperform DBC, there were indications from the profiling pointing to fetch latency from the database to the application. As we established earlier, a file server will usually be run with some form of database to keep metadata. It would therefore be interesting to see if the Chunk approach would see more similar results to DBC. This could also be done by measuring the CPU time of the application when running the solutions, which preempts time spent waiting for other processes.

5.1.2 Chunk storage in RDBMS

Our approach using a document-oriented database is replicable using a relational database. It could be interesting to see whether relational databases saw improved runtime performance and memory utilization compared to document-oriented ones.

5.1.3 Break-even points for chunk sizes

As was seen in the results, varying the size of chunks does impact performance. This may relate to what hardware is used, the file system of the operating system, or (for the DBC approach) the DBMS. Although the performance differences were not particularly large, only two chunk sizes were tested. Testing a broader span of chunk sizes could lead to further decreases in execution times.

Bibliography

[1] "Glossary: Confidentiality." (2023), [Online]. Available: https://csrc.nist.gov/glossary/term/confidentiality (visited on 06/01/2023).

[2] "Glossary: Cryptography." (2023), [Online]. Available: https://csrc.nist.gov/glossary/term/cryptography (visited on 06/01/2023).

[3] "Glossary: Encryption." (2023), [Online]. Available: https://csrc.nist.gov/glossary/term/encryption (visited on 06/01/2023).

[4] "Glossary: Decryption." (2023), [Online]. Available: https://csrc.nist.gov/glossary/term/decryption (visited on 06/01/2023).

[5] "Glossary: Encryption algorithm." (2023), [Online]. Available: https://csrc.nist.gov/glossary/term/encryption_algorithm (visited on 06/01/2023).

[6] "Glossary: Cryptographic key." (2023), [Online]. Available: https://csrc.nist.gov/glossary/term/cryptographic_key (visited on 06/01/2023).

[7] "Glossary: Cryptanalysis." (2020), [Online]. Available: https://csrc.nist.gov/glossary/term/cryptanalysis (visited on 06/01/2023).

[8] "Glossary: Asymmetric cryptography." (2020), [Online]. Available: https://csrc.nist.gov/glossary/term/asymmetric_cryptography (visited on 06/01/2023).

[9] "Glossary: Symmetric cryptography." (2020), [Online]. Available: https://csrc.nist.gov/glossary/term/symmetric_cryptography (visited on 06/01/2023).

[10] B. Kaliski, "The mathematics of the rsa cryptosystem," 2006. [Online]. Available: https://www.nku.edu/~christensen/the%5C%20mathematics%5C%20of%5C%20the%5C%20RSA%5C%20cryptosystem.pdf.

[11] W. Commons, *File:simple symmetric encryption.png — wikimedia commons, the free media repository*, [Online; accessed 13-March-2023], 2023. [Online]. Available: https://commons.wikimedia.org/w/index.php?title=File:Simple_symmetric_encryption.png&oldid=738050725.

[12] "Glossary: Block cipher." (2020), [Online]. Available: https://csrc.nist.gov/glossary/term/block_cipher (visited on 06/01/2023).

[13] "What is a stream cipher?" (1998), [Online]. Available: http://security.nknu.edu.tw/crypto/faq/html/2-1-5.html (visited on 06/01/2023).

[14] "Glossary: Key stream." (2022), [Online]. Available: https://csrc.nist.gov/glossary/term/key_stream (visited on 06/01/2023).

[15] M. Dworkin, E. Barker, J. Nechvatal, *et al.*, "Advanced encryption standard (aes)," en, National Institute of Standards and Technology, Tech. Rep., 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197.

[16] "Data encryption standard (des)," National Institute of Standards and Technology, Tech. Rep., 1999. DOI: hhttps://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf.

[17] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, *Twofish: A 128-bit block cipher*, 1998. DOI: https://www.schneier.com/wp-content/uploads/2016/02/paper-twofish-paper.pdf.

[18] "Glossary: Integriry." (2022), [Online]. Available: https://csrc.nist.gov/glossary/term/integrity (visited on 06/01/2023).

[19] M. Dworkin, "Recommendation for block cipher modes of operation," National Institute of Standards and Technology, Tech. Rep., 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197.

[20] "Glossary: Initialization vector." (2022), [Online]. Available: https://csrc.nist.gov/glossary/term/initialization_vector (visited on 06/01/2023).

[21] W. Commons, *File:cbc encryption.svg — wikimedia commons, the free media repository*, 2020. [Online]. Available: https://commons.wikimedia.org/w/index.php?title=File:CBC_encryption.svg&oldid=411984531 (visited on 04/2023).

[22] W. Commons, *File:cbc decryption.svg — wikimedia commons, the free media repository*, 2020. [Online]. Available: https://commons.wikimedia.org/w/index.php?title=File:CBC_decryption.svg&oldid=411984685 (visited on 04/2023).

[23] G. J. Simmons, "Symmetric and asymmetric encryption," *ACM Comput. Surv.*, vol. 11, no. 4, pp. 305–330, 1979, ISSN: 0360-0300. DOI: 10.1145/356789.356793. [Online]. Available: https://doi.org/10.1145/356789.356793.

[24] J. M. Pérez. "Progressive image rendering." (), [Online]. Available: https://jmperezperez.com/blog/render-conf-oxford-2017/.

[25] Mozilla. "Web video codec guide." (), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Video_codecs.

[26] "Youtube recommended upload encoding settings." (), [Online]. Available: https://support.google.com/youtube/answer/1722171?hl=en#zippy=%5C%2Ccolor-space%5C%2Cvideo-resolution-and-aspect-ratio%5C%2Cbitrate (visited on 03/23/2023).

[27] A. e. a. Weeks. "Linux system administrators guide." (2004), [Online]. Available: https://tldp.org/LDP/sag/html/filesystems.html.

[28] "File systems driver design guide." (Sep. 2022), [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/.

[29] R. Nordvik, K. Porter, F. Toolan, S. Axelsson, and K. Franke, "Generic metadata time carving," *Forensic Science International: Digital Investigation*, vol. 33, p. 301 005, 2020, ISSN: 2666-2817. DOI: https://doi.org/10.1016/j.fsidi.2020.301005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S266628%201720302547.

[30] M. Ray *et al.* "Filestream (sql server)." (Feb. 2023), [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/blob/filestream-sql-server?view=sql-server-ver16.

[31] "File server: Definition and basics." (2023), [Online]. Available: https://www.ionos.com/digitalguide/server/know-how/file-server/ (visited on 03/23/2023).

[32] "What is a relational database (rdbms)?" (2023), [Online]. Available: https://www.oracle.com/in/database/what-is-a-relational-database/ (visited on 04/20/2023).

[33] H. Vera, W. Boaventura, M. Holanda, V. Guimaraes, and F. Hondo, "Data modeling for nosql document-oriented databases," in *CEUR Workshop Proceedings*, vol. 1478, 2015, pp. 129–135.

[34] E. Bisong and E. Bisong, "An overview of google cloud platform services," *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*, pp. 7–10, 2019.

[35] J. Varia, S. Mathew, *et al.*, "Overview of amazon web services," *Amazon Web Services*, vol. 105, 2014.

[36] Personopplysningsloven, *Lov om behandling av personopplysninger (personopplysningsloven)*. Lovdata, 2018, vol. LOV-2018-06-15-38. [Online]. Available: https://lovdata.no/dokument/NL/lov/2018-06-15-38.

[37] Mozilla. "Time to first byte." (2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Time_to_first_byte.

[38] J. Anderson. "Memory paging." (Jan. 2023), [Online]. Available: https://www.techtarget.com/searchitoperations/definition/memory-paging (visited on 05/04/2023).

[39] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988, ISSN: 0001-0782. DOI: 10.1145/42411.42415. [Online]. Available: https://doi.org/10.1145/42411.42415.

[40] L. Null and J. Lobur, *The essentials of computer organization and architecture (2. ed.)*. Jan. 2006, p. 315, ISBN: 978-0-7637-3769-6.

[41] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. CreateSpace Independent Publishing Platform, 2018, ISBN: 198508659X.

[42] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," New York, NY, USA: Association for Computing Machinery, 2009, ISBN: 9781605586236. DOI: 10.1145/1534530.1534544. [Online]. Available: https://doi.org/10.1145/1534530.1534544.

[43] W. Bugden and A. Alahmar, "Rust: The programming language for safety and performance," Jun. 2022. DOI: 10.48550/arXiv.2206.05503.

[44] E. Yepis, "Comparing tag trends with our most loved programming languages," Jan. 2023. [Online]. Available: https://stackoverflow.blog/2023/01/26/comparing-tag-trends-with-our-most-loved-programming-languages/.

[45] G. Barany, "Python interpreter performance deconstructed," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2014. DOI: 10.1145/2617548.2617552.

[46] M. Vaidehi and B. J. Rabi, "Design and analysis of aes-cbc mode for high security applications," in *Second International Conference on Current Trends In Engineering and Technology - ICCTET 2014*, 2014, pp. 499–502. DOI: 10.1109/ICCTET.2014.6966347.

[47] "Time — time access and conversions." (Aug. 2022), [Online]. Available: https://docs.python.org/3.10/library/time.html (visited on 02/25/2023).

[48] "The python profilers." (Aug. 2022), [Online]. Available: https://docs.python.org/3.10/library/profile.html (visited on 02/16/2023).

[49] "Pycryptodome." (Jan. 2023), [Online]. Available: https://pycryptodome.readthedocs.io/ (visited on 02/14/2023).

[50] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, pp. 97–113, Jun. 2003. DOI: 10.1145/857076.857077.

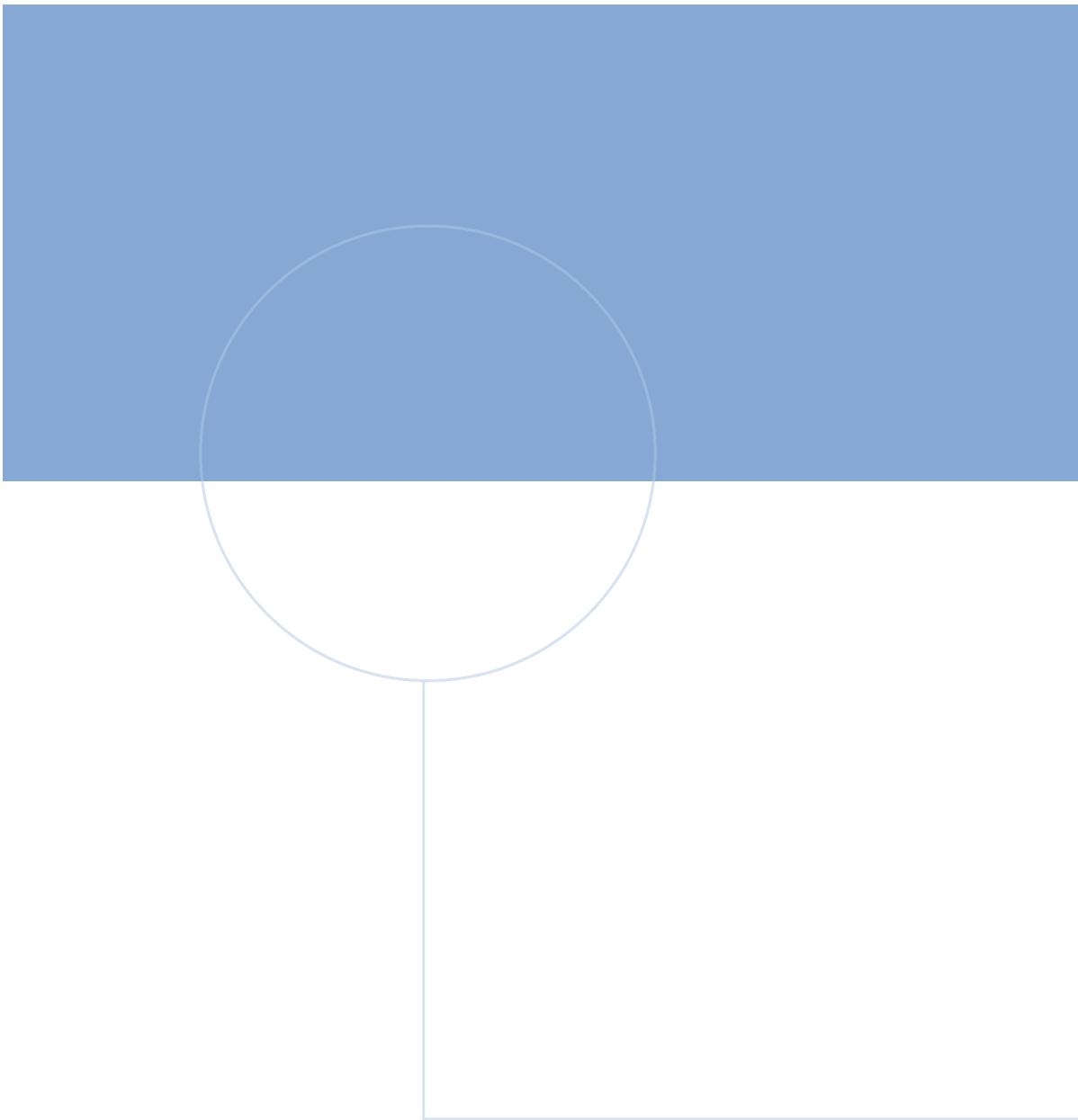
[51] "Unittest — unit testing framework." (Aug. 2022), [Online]. Available: https://docs.python.org/3.10/library/unittest.html (visited on 02/16/2023).

[52] "Tracemalloc — trace memory allocations." (Aug. 2022), [Online]. Available: https://docs.python.org/3.10/library/tracemalloc.html (visited on 04/20/2023).

[53] R. Sears, C. Ingen, and J. Gray, "To blob or not to blob: Large object storage in a database or a filesystem?" *Computing Research Repository - CORR*, Jan. 2007.

[54] "What is powershell?" (Oct. 2022), [Online]. Available: https://learn.microsoft.com/en-us/powershell/scripting/overview?view=powershell-5.1.

[55] S. Josefsson. "The base16, base32, and base64 data encodings." (Oct. 2006), [Online]. Available: https://www.rfc-editor.org/rfc/rfc4648.



NTNU

Norwegian University of
Science and Technology