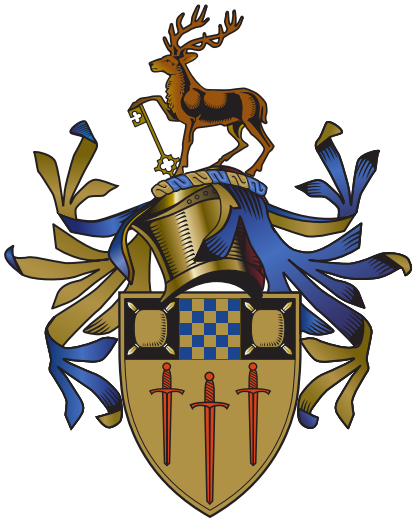

Attribute-Based Authorisation as a Micro-service with Dynamic Encryption Selection

A dissertation submitted in partial fulfilment of
the requirements for the award of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

by

Aran Simon Jannson

6751704

May 14, 2025

Department Of Computer Science
University of Surrey
Guildford, England
United Kingdom
GU2 7XH

Supervised By: Dr Nick Frymann

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

Aran Simon Jannson
May 14, 2025

Abstract

This project provides an Attribute-Based Access Control (ABAC) authorisation system designed for low end devices with limited resources. Users can securely share files using a token based sharing scheme, where these tokens include access permissions and an access time frame. The system is implemented using a microservice architecture to ensure lightweight, scalable performance on a large number of hardware levels. The system dynamically selects an encryption profile based on the hardware capabilities of the device, in order to provide more secure algorithms when system resources allow it. More efficient profiles are chosen when the hardware of the systems is capable of processing the requirements efficiently. A web application demo is used to provide the system, featuring an interface that enables file navigation, access, and sharing through QR codes or shareable links. This project highlights how dynamic encryption profiles, combined with ABAC permission tokens, can provide secure and efficient access control even on resource limited systems.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr Nick Frymann, for his invaluable support and insightful guidance throughout the duration of this project. His feedback has been an important part in shaping the direction of my work and helping me explore approaches that I might not have otherwise considered.

Abbreviations

MSA	Microservice Architectures
SAAS	Software As A Service
AC	Access Controls
ABAC	Attribute Based Access Control
RBAC	Role Based Access Control
OAuth	Open Authorisation
ZTA	Zero Trust Architecture
JWT	JSON Web Tokens
KDF	Key Derivation Function
JTI	JSON Web Token Identifier
JWE	JSON Web Encryption
AES	Advanced Encryption Scheme
CBC	Cipher Block Chaining
MAC	Message Authentication Code
CCM	CBC-MAC

Contents

List of Tables

List of Figures

1 Introduction

1.1 Overview

Secure file sharing is an essential component of modern digital platforms. However, where authorisation management and access control clients are concerned, typical implementations use a large amount of system resources leading to an increase in minimum system requirements as time passes. This presents challenges in providing secure and cost-effective authorisation management platform. This project will aim to solve this issue by tuning the security schema based on the system resources of the system it is being hosted on, so that an efficient and secure solution can be provided for a lightweight low resource system to achieve this goal. However, this task is not simple as balancing efficiency and security can be challenging.

Authorisation management is a necessary component of a secure system to ensure access rights to resources that are controlled by the owner of the data, and are accessed by the intended user(s) only. Typical implementations provide ways of creating rules to apply these controls. By achieving this, the data is securely shared to a secondary user with appropriate permissions such as operation and time-based restrictions. This project will achieve this using secure sharing methods through token sharing, similar to Auth0 [3] and Keycloak [4]. It will provide users access to files and resources using this method. However, the benefits of this project will be provided by the scalability of the system, as well as the encryption schemes used. This level of scalability will be a challenge to accomplish as pre-defined schemes will need to have a range of different encryption levels. When providing these profiles having the correct set of parameters will be a major task in succeeding with this project.

This system has a strong use case as a file sharing service, targeting personal and enterprise users to achieve a cost-effective solution. This project aims to offer a solution to the problems outlined above in a package that can be hosted on devices with low system resources, providing the functionality through a RESTful API allowing integration in multiple environments. This will be paired with a web interface to control the file-sharing aspects functions on a per-user basis.

1.2 Project Goals

The project aims to deliver an easy-to-integrate and efficient system suitable for a wide range of hardware profiles. Specific objectives include:

Key system components will need to include:

- A token-based file sharing mechanism embedding permissions and a defined expiration time.
- A dynamic encryption algorithm selector based on system CPU, memory, and system load.
- Creation of, or integration with, a scalable file storage solution.
- A web frontend management dashboard enabling users to control file sharing and related functions.
- A deployable set of functions to various systems.

Objective	Description
1	Research ABAC and microservice systems.
2	Develop an ABAC platform with a functional file navigator.
3	Investigate JWT security and supported security features.
4	Implement file sharing through QR codes and generated links.
5	Integrate with S3 bucket storage objects.
6	Design and implement the receiver-side sharing mechanism, as well as other ABAC permission attributes.
7	Benchmark different encryption schemes under varying system conditions.
8	Enable encryption scaling based on available hardware resources.
9	Provide a UI that displays all files shared by and to the logged in user.
10	Enable Offline availability through a mobile application.
11	Evaluate additional security measures that can be taken.

Table 1: Project Objectives

1.3 Project Business Possibilities

The final product aims to be able to provide companies as well as individuals an affordable file authorisation solution with a simple to use MSA based service using RESTful API requests. The system aims to be cost effective, allowing companies to run it on low-performance hardware and allow businesses to save on spending large amounts on authorisation systems that are integrated into a traditional monolithic system and also save on investing in high end hardware to achieve this goal. Using this, businesses can connect to a authorisation system that is cheaper to maintain on a secondary device. Calls can be made to the device running the microservice in order to grant access to content.

1.4 Initial Plan

The initial plan for the project was structured into several sections to systematically develop and validate the Attribute-Based Access Control (ABAC) microservice system.

The first section involved researching and understanding the specific challenges associated with building secure authorisation systems for resource-constrained environments. This included searching for similar platforms that provide one or all features such as an ABAC model, various encryption standards, tokenization techniques, and scalability considerations within microservice architectures.

Following having researched into the requirements of the platform, implantation can be started. The high-level system architecture is planned, defining the core components that make up a lightweight ABAC authorisation microservice using Express Js [9], a secure file storage solution using Supabase S3 Buckets [49], and a responsive frontend application built with Next.js

15 [14]. Considerations were made about how the microservice would dynamically select encryption schemes based on hardware analysis and performance metrics, balancing security and performance to the best extent possible.

Development was planned to proceed in a modular fashion:

- **Backend Implementation**: Create a core microservice responsible for token generation, validation, selection of encryption scheme, and clarification of user permission.
- **Frontend Development**: Build a functional web interface that allows users to upload, manage, and share files, as well as interact with a selector for which ABAC permission settings can be applied to a token.
- **System Integration**: Connect the frontend application to the microservice through API calls, ensuring functional and secure handling of tokens and access control logic.

A performance testing function is designed to simulate different encryption profiles. This involves taking advantage of CPU, memory, and system load limits in test environments to check what the highest profile level that can be applied to the current hardware stack being used is.

Testing scenarios were set out in two stages:

- **Functional Testing**: Verify that all core features, such as token-based file sharing, QR code generation, and permission enforcement function as intended in multiple scenarios. Such as valid and invalid token generation and usage.
- **Performance Testing**: Benchmark token generation latencies, successful and unsuccessful file access across various simulated scenarios and hardware stacks. Deciding on the pre-set profiles for various levels of hardware based on executed tests on emulated and real-world hardware.

The initial plan also included the development of demonstration features, such as:

- Generating QR codes and shareable links from the frontend based on the generated token.
- Providing error messages and fallback mechanisms when hardware was insufficient for certain security levels. Such as choosing a lower level of encryption for the secret key generation.

Throughout the project timeline, Docker [8] was chosen as the best deployment platform for the implementation of a microservice architecture. Allowing the microservice and supporting components to be containerised for consistency between test sets and future scalability. Platforms such as Apptainer [2] were also explored in order to provide a containerisation platform that uses fewer resources itself. However, due to the seamless implementation of Docker with the tech stack being used, it was chosen over Apptainer.

This structured and sectioned initial plan ensured that both the technical requirements of the ABAC system and the usability goals for the demonstration were taken into account.

2 Literature Review

In this literature review I will be exploring Access controls (AC), micro-services and storage solutions for an authorisation platform. In order to provide an efficient and reliable solution for an authorisation microservice with scalable encryption schemes based on system resources. In addition to this I will be exploring various types of access controls as well as authentication schemes in order to find the best solution.

2.1 Attribute Based Access Control

2.1.1 Access Controls

Access control (AC) systems provide a security framework to apply rule sets to authorisation platforms. The system is in charge of how a user is provided a permission set which is required for the user to be authorised in order to access a specified resource or platform [41,52]. There are various types of AC systems for different use cases. This project focuses on ABAC with a brief over view of RBAC which offers an attribute-based approach as well as a role-based approach. Both methodologies provide different benefits with ABAC being a Discretionary model whilst RBAC provides a Non-Discretionary model to authorisation.

2.1.2 Different Types Of Access Controls

Different types of AC systems involve ABAC, RBAC, Mandatory Access Control (MAC) , Discretionary Access Control (DAC), Non-Discretionary Access Control (NDAC) [33,41,44] and more. Each of these provide a different approach to an AC system.

Access Control Model	**Description**
Role-Based Access Control (RBAC)	Access is granted based on a user's role (e.g., Admin, Manager).
Attribute-Based Access Control (ABAC)	Access is granted based on predefined attributes (e.g., Time, Read/Write Permissions, Location).
Discretionary Access Control (DAC)	Access is set by the owner of the data.
Non-Discretionary Access Control (NDAC)	Access is set by the company in charge of the data.
Mandatory Access Control (MAC)	Access is enforced by a centralized authority with strict policies in use, often used for high security systems (e.g., military).

Table 2: Access Control Type Examples

2.1.3 ABAC

ABAC is an access control framework which uses attributes that have been assigned to it to provide authorisation. ABAC provides a strong foundation on which to authorise users. Microservices can take advantage of ABAC in order to provide a flexible platform for assigning users access to specific systems or data.

ABAC can follow either a DAC or an NDAC approach depending on the use case. Following a NDAC approach to ABAC would provide a service which allows business owners to apply attributes to their employees in order to allow access to systems or data. For the scope of this dissertation, I will be following a DAC approach which allows each user autonomy over their own data.

2.2 Challenges of ABAC

Since the system provides individuals in each specific case with a unique set of permission and rules, it can be challenging to process this information on a large scale deployment. In order to solve this issue I will be taking advantage of JSON Web Tokens (JWT) in order to provide a token based approach to storing the necessary rule sets. A JWT includes fields such as user identity, permitted duration, permission set such as **Read** and **Write** along side any other field required [46]. This JWT can be verified using an authorisation server which can be placed as an additional secondary MSA server or be included in the core system itself [46].

2.2.1 JSON Web Tokens (JWT) & Encryption Algorithm Types

Another challenge faced is encrypting the JWT as well as the secret key. This issue does not only span the task of having an encrypted token and secure secret keys provided, but also the algorithm that can be used in order to accomplish this task on a variety of hardware levels.

JWT Algorithms	**Type**	**Description**
PS256	Asymmetric	RSASSA-PSS using SHA-256
RS256	Asymmetric	RSASSA-PKCS1-v1_5 using SHA-256
ES256	Asymmetric	ECDSA using P-256 and SHA-256
EdDSA	Asymmetric	Edwards-Curve Digital Signature Algorithm
HS256	Symmetric	HMAC using SHA-256

Table 3: JWT Algorithms and Their Characteristics In JWT [26]

2.2.2 JWT Format and How It Works

JWT can be used as a compact URL safe form of transferring a JSON package which is encoded using a JSON Web Signature (JWS) [34, 36]. The JSON object, known as the JWT Claim Set, is formed of value pairs known as a claim value that represent the claims.This claim can hold non-standard JSON format data [36]. JWS is used on the payload in order to sign the JWT generated and is signed using a Message Authentication Code (MAC). The encryption keys referenced above are applied to generate a signature for the JWT, combining a hash function such as SHA-256 with a secret key. The recipient can confirm the token using the same secret key that was used in the signature.

2.2.3 Advanced Encryption Standard (AES)

JWT tokens can be encrypted using the Advanced Encryption Standard (AES) based encryption algorithms. AES is a symmetric block cipher that has become the standard for securing

digital communications. The block size is 128 bits and is standard on all versions. It supports a variety of key sizes which include 128, 192 and 256 bit's respectively. Since it is symmetric, the same key is used for encryption and decryption. Given that the same key is provided, the input value (encrypted value) will provide the same data that was present before encryption when decrypted. Multiple versions of AES exist, including AES-128, AES-192 and AES-256 based on the key length used [23].

AES has withstood, extensive crypto analysis. While attacks such as biclique crypto analysis [23] slightly reduce the computational complexity of brute force they still remain impractical for real world usage. The design of AES has shown strong resilience to linear and differential crypto analysis [23]. The secure nature of AES is derived from the mathematical strength of the S-box.

2.2.4 AES-CCM

AES has a flaw with the lack of built-in authentication, to solve this, modes such as CBC-MAC (CCM) and GCM are implemented to provide integrity and authenticity. In CCM a message authentication code (MAC) is taken advantage of to apply authentication to encrypted data [23, 31]. A MAC is a type of authentication string that is generated from a message and a secret key, it is used to prove that the encrypted data has not been altered during transmission. When applying a CCM, a CBC-MAC is first applied to the data and plaintext, after which it outputs a MAC variable. Then, using a counter mode, the plaintext and the MAC are encrypted. In order to decrypt an AES scheme with CCM, a counter mode is used to decrypt the received data, and the CBC-MAC is recomputed to verify authenticity. In order for this to succeed, the same key must be used during encryption as well as decryption, otherwise the MAC can not be successfully recovered due to the failure to authenticate the data [31].

2.2.5 AES-GCM

AES-GCM (Galois/Counter Mode) is a widely adopted authenticated encryption mode that provides both confidentiality and integrity. It combines AES in counter mode with a Galois field-based authentication mechanism [32], allowing for high performance and secure implementations. Its efficiency is noticeable on hardware that supports ARMv8 crypto extensions, which makes it ideal for systems that handle large amounts of data or require efficient encryption [32]. Due to its support for ARMv8 it is ideal for microservices that run Linux based operating systems.

2.2.6 Key Derivation Function (KDF)

A Key Derivation Function (KDF) provides a function in which you can take basic strings such as passphrases and create secure cryptographic keys using them. In the KDF family, the PBKDF2 (Password-Based Key Derivation Function 2) remains one of the most widely adopted standards. PBKDF2 provides enhanced security for the generated key by using a pseudo-random function [10, 11] such as HMAC-SHA-1 and HMAC-SHA-2 [11] combined with a salt value to add some randomness. This enables brute force attacks from being less likely to succeed. The strongest feature of PBKDF2 is its adjustable iteration count, which controls the computational cost of the key derivation by modifying the number of times the digest is applied. These digests are SHA-256, SHA-384 and SHA-512 making brute-force attacks impractical the higher the iteration count is.

2.3 Microservice Architecture (MSA)

Microservice Architectures (MSA) are a standard in which systems are split into smaller sub-systems placed on multiple servers. MSA has gained significant traction in recent years after the success of Service-Oriented Architecture's [39]. MSA based systems are usually accessed using RESTful API requests. MSA's provide a scalable structure, due to the systems being designed to only process a set section of the platforms feature set. Dedicated MSA systems can perform specific tasks associated with the system, allowing for a more efficient use of system resources when used correctly.

However, one of the downside's of MSA include the fact that each microservice has to have its own set of security protocols included [29], such that a streamlined universal security system cannot be implemented. Each system would have its own level of resources, meaning that a system with lower-quality hardware could face an issue processing a high level of encryption that another part of the system might not struggle to process. Thus, unique security systems must be implemented for each MSA based system in use, whilst they all work in a parallel manner to each other in order to provide a complete functionality.

The growing popularity of microservices can be attributed to the difficulties of maintaining large applications over time [51]. Companies have migrated to cloud-based systems and SAAS solutions over the years in order to reduce the strain on their local machines so that it may provide a more efficient service. These systems tend to use MSAs due to traditional architectures being insufficient in the modern world [51]. They have an ability to split an application not only saving on system resources but in addition being more cost effective in the long term. Evidence shows that most organizations moving towards an MSA start with a monolithic architecture [51]. This is where a system is run on a single machine/server providing all services in one location as a unified process and later move on to a MSA model. Figure 1 below is an example of an MSA based system [42].

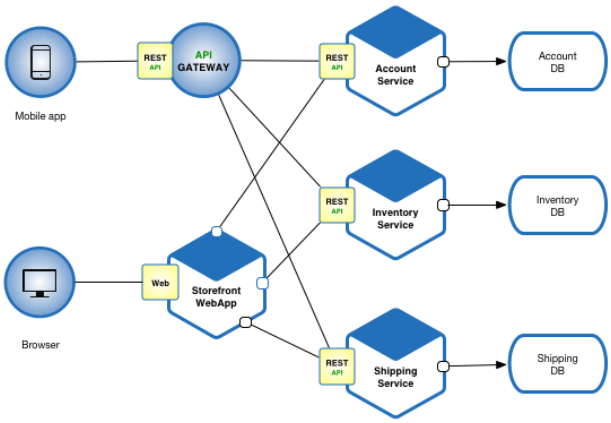

Figure 1: MSA Example [42]

In addition an MSA system can also allow a company unlimited freedom regarding what tools are used in each system. For example one part of the system on one of the many MSA servers could be running a Python based service whilst another MSA system which communicates with this system could be running on JavaScript. MSA systems often utilise lightweight stacks such as

embedded Jetty or SimpleWeb [51]. Given that the systems are designed to be able to effectively communicate there is no restrictions on which languages or tools can be used on any of the MSA systems in use.

MSA also comes with the additional responsibility of managing multiple platforms. This could lead to more possibilities of failure and or faults in a system, as well as additional security vulnerabilities if one system gets compromised. So when considering the usage of an MSA based platform the benefits must be weighed against the risks along side the reason for considering implementing such a system. It also ensures that the best security practices possible are implemented in order to have a safe user experience.

2.3.1 ABAC In MSA

An ABAC system can be implemented using containerisation platforms such as Docker. Using a multi-platform structure, we can separate the back-end platform from the front-end enabling the ability to have the ABAC authorisation service in a separate container from the user interface, file manager and any additional services [45]. The containers can all be run on separate hardware stacks as required, saving the resources required to maintain the service. Connecting all back-end components using a RESTful API. These containers can provide an environment for the required service that is tuned for the specified use case, allowing for an optimised experience with simplified scalability as needed [45]. The authorisation request will be passed through to the container, and then a response will be returned accordingly.

2.4 Fundamentals Of Authentication For Authorisation

Authentication is a fundamental security feature used in modern day systems. It provides a way to verify a user's identity information such as name, age and contact details. It ensures that the parties involved in an interaction are who they claim to be, providing a system confidence to allow access to interact with the programme [28]. Authentication typically works by combining user credentials, such as an email and password, biometric data and passkeys combined with a protocol to enable trust between the user and the system.

Creating an authentication system is not easy. Effective authentication does not only prevent unauthorised access but also enables the ability to provide authorisation, encryption, and secure session management. There are two main approaches to implementing authentication, symmetric-key authentication and public key based user authentication.

2.4.1 Symmetric Key Authentication

Symmetric key user authentication relies on a shared secret key between two systems, such as a user and a server. Beforehand a shared key is established, the key must remain secret and only known by the two parties involved. A random challenge is generated by the server which is forwarded on to the user. The user then encrypts this challenge using the shared key agreed on earlier and then returns this encrypted value to the server [35]. The user is then verified when the server can decrypt the encrypted challenge with the same shared key.

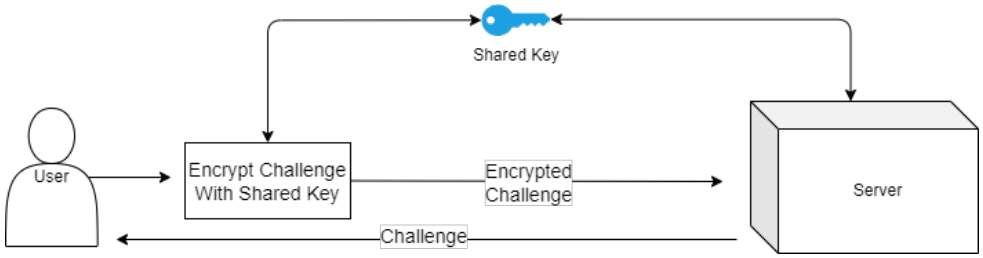

Figure 2: Symmetric Key Authentication Illustration

Symmetric key authentication is computationally less intensive than the public key alternatives. While it provides an efficient and secure authentication system, it struggles on larger systems due to the need to track a vast amount of shared keys, which would require significant system resources. Symmetric key authentication is widely used for authentication servers such as Kerberos [37] which has the sole task of distributing session keys for secure communication between two parties.

2.4.2 Public Key Authentication

Public key based user authentication is based on the use of a pair of keys, a public and private key generated for each entity. These keys are used to verify a user without requiring a traditional password, eliminating risks associated with these passwords [30].

Public Key Authentication works by providing the server with the users public key to start with. Anyone with access to this public key can encrypt data with it and the user can decrypt and read the data [16]. The private key is kept secret by the user and is used as a form of identifier to prove who they are to the server. Proving the user with a challenge similar to symmetric key authentication. The user encrypts this challenge with their private key creating a unique signature, the server can then verify the user using the stored public key. [16, 30]

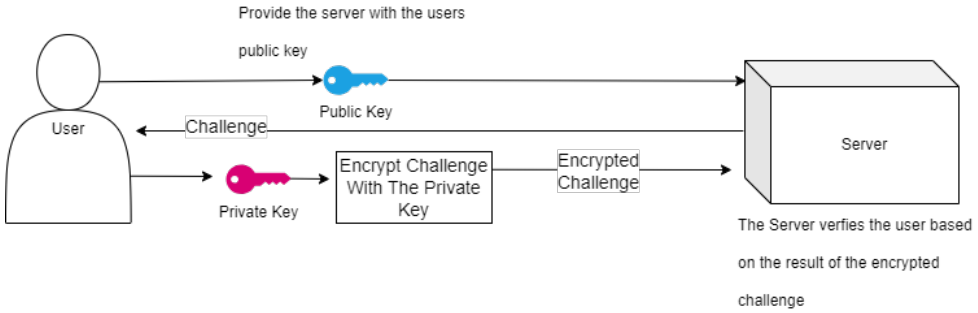

Figure 3: Public Key Authentication Illustration

2.5 Authorisation

Authorisation systems provide platforms with a tool that can grant users access based on specified requirements. The system determines what a user has access to, the level of access they

have and or if they are allowed to view the data at all. Only if these prerequisite are fulfilled will it allow access to be granted to the user. Authentication platforms verify a user's identity to prove they are who they say they are. An authorisation service, however, assumes authentication has already taken place. Rather than verifying identity, it determines whether access to specified data is permitted. This is accomplished following a Zero Trust Architecture (ZTA) structure.

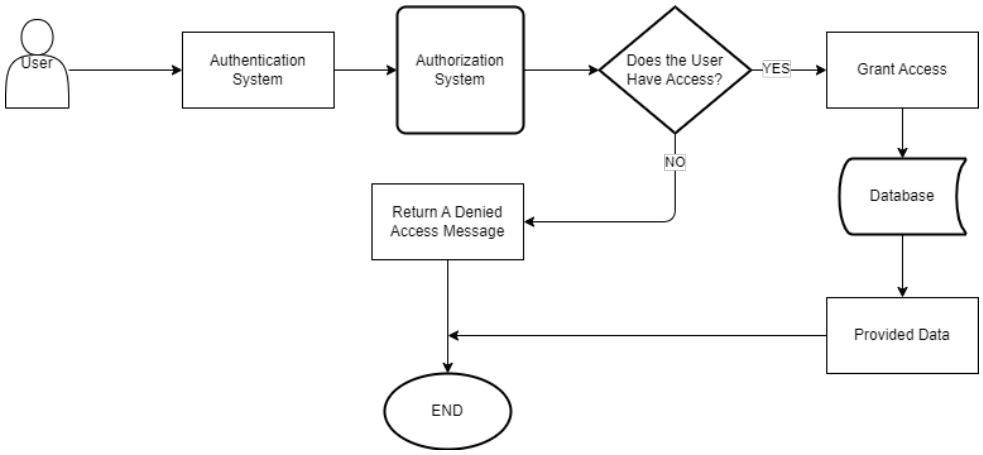

Figure 4: Basic Illustration Of An authorisation System

2.5.1 Zero Trust Architecture (ZTA)

Authorisation services use a Zero Trust Architecture (ZTA) approach meaning the system presumes no one has access unless specifically specified otherwise. This is done by enforcing strict policies, applying the lowest level of privileges possible [43]. After this, it begins deciding on what access will be allowed based on factors such as a user's identity, role, location or time. For example a user may only be granted access as an administrator thus everyone else having there access revoked if they try to access data which is flagged as administrator only. It performs a user's access check each time they try to access restricted data or systems in order to load the latest list of privileges for an authenticated user. This is done to check if any changes are made [43]. The system also allows for a mitigation of threats due to enforcing the lowest level of privileges first rule. Aligning with data protection rules as sensitive data would be automatically blocked from being accessed unless otherwise specified.

2.5.2 OAuth

OAuth is a protocol designed to implement authorisation in applications using a token-based system. OAuth was invented to allow limited authorisation unlike permanent access which allows authentication to be provided until removed in its entirety. OAuth would allow a user access specifically based on tokens generated with permission sets included in order to provide access to exactly what is required and no more [38]. The token such as a JWT would provide limited authorisation for a user based on pre-defined parameters within the token. The benefits of OAuth include increased security that prevents user credentials from being exposed [38]. The challenges in OAuth involves ensuring proper use of the included JWT. The system must ensure that the token is securely formatted and shared without the possibilities of unauthorised access being granted using the data provided by the token [38]. An OAuth service would include an

authorisation server, resource server, access token and owner along side a front end client in order to function.

The service would function when the client requests data from the resource server (e.g. user's contact details). The authorisation server would then query whether to grant access to this information or not from the owner of the data. Provided authorisation is granted, an authorisation token is generated. This token is then forwarded onto the requesting client which validates the token with the authorisation server using an access code. Provided that the access token is accepted, the client then issues the requesting user with their request. Providing the data that was queried [38].

2.6 Buckets

Buckets provide a containerised solution to storing data. Allowing for content-specific partitioning. They allow for a more controlled storage solution that allows each bucket to have its own set of default access rights applied uniquely in comparison with the majority of the rest of the system [1]. Private buckets provide a Row-Level Security-Based access, allowing for a perfect set-up for confidential data access as it insures only the attended audience can have access to the required data [49]. Although public buckets provide an easier but less secure solution, public buckets void all forms of authorisation by passing implemented pre defined AC systems [49]. Leaving the system vulnerable to possible unintended access. Examples of Bucket storage providers could be Supabase Bucket Storage system paired with there PostgreSQL database [49] and Amazon S3 Buckets [25, 27].

2.6.1 Benefits Of Buckets

Bucket storage benefits organisations due to the cost effectiveness of running them. Rather than paying for a large-scale storage server. Companies can have multiple buckets in which they can scale as required, providing more storage only when needed. Buckets also benefit ABAC implementations. Being able to provide JWT tokens to authorise a user for a specified bucket only. In addition it provides security for other individual's data as the token only authorises access to the user for that one specific bucket. The buckets do not have to be on one server or with one service provider. Providing access to data across multiple sites is possible, assuming a global authentication system is in place to verify users attempting to access data from different locations. [1, 25, 27].

2.6.2 Challenges Of Using Buckets

The challenges of Bucket storage revolves around how they get implemented. If a token generated to authorise a user is not formatted or secured effectively enough, the individual could forward this token to someone else allowing an unauthorised user from gaining access to a bucket [24, 50]. This causes security breaches to occur. Buckets may also need to comply with different privacy laws, such as GDPR, depending on the country where they are hosted. Forcing companies to implement one or more privacy laws into their system [13].

2.7 Conclusion

The suitability and strength of an Attribute Based Access Control (ABAC) for scalable and secure authorisation systems, particularly when implemented with a microservice architecture, are

explored and analysed. ABACs ability to set access policies using flexible and specific attributes makes it the perfect choice for dynamic environments where user roles and contexts frequently change. It provides a system for when each user has their own specific set of access rights based on third-party factors.

The benefits of integrating ABAC with JSON Web Token (JWT) based token systems address the challenge of sharing authorisation data efficiently and securely across multiple instances of the platform, as well as different components of the system. By including permission sets and data directly into cryptographically signed tokens, ABAC can be operationalised without constant state checks, allowing for improved scalability and performance on low end platforms.

Encryption is a vital element in securing tokens and making sure authorisation applications are reliably executed. The use of symmetric encryption algorithms, such as AES and its authenticated versions such as AES-CCM, provides a balance of performance and security, when applied using a secret key application using Key Derivation Functions (KDF) such as PBKDF2. This encryption style ensures that systems with multiple levels of system resources can still provide strong security levels.

Microservice Architectures (MSA) can be used to complement an ABAC platform by providing a modular environment for an authorisation system. However, they introduce challenges of their own in terms of security consistency, especially when different services run on different levels of hardware stacks. Containerisation tools like Docker are useful in helping with these challenges by enabling environment-specific configurations in which the system can provide a set resource layout from the resource pool of the hardware it is running on.

In addition, authentication through both symmetric and asymmetric models shows the importance of identity verification before being able to apply authorisation rules. The inclusion of protocols like OAuth and ZTA reinforce the need for specific and secure, lowest-privilege first access control models that adapt to unauthorised access attempts.

Finally, bucket based storage systems show their suitable compatibility with ABAC model systems. Buckets, particularly those offering Row-Level Security, provide effective and scalable means of storing sensitive data for each individual user whilst enabling pre-defined access control rules. However, the security of buckets is based on the implementation being applied making sure that the correct use and management of authorisation tokens are used.

In summary, the convergence of ABAC, JWT, adaptive encryption, MSA, and secure storage solutions presents a powerful bundle of methodologies to be able to build a secure and scalable access control platform. These technologies, when integrated correctly, provide a strong base for the ABAC microservice with dynamic encryption selection to function successfully.

3 Legal, Social, Ethical & Professional Issues

3.1 GDPR

The project provides a file sharing platform which stores and accesses user data that has been uploaded to the S3 Buckets. The project follows the UK GDPR [19] guidelines in order to protect user data to the best level possible. The following considerations were made during development of the project:

- All files are encrypted and decrypted on the server side, and access is restricted using ABAC policies embedded in JWT's.
- JWT's include expiry times and unique JTIs, which help enforce limited-time and single-use access, reducing the risk of misuse.
- Supabase S3 buckets are configured with strict access control; file access is validated against a permissions registry stored in the Supabase database.
- Tokens use dynamically selected encryption schemes set accordingly based on the host system's performance and hardware, prioritising stronger encryption when resources allow it.

3.2 Ethical Considerations

Security is significant when it comes to ethics, especially since this system could be misused for unauthorised access to user data if improperly designed. The following steps mitigate these risks:

- Encryption and decryption are applied exclusively on the microservice, making sure sensitive data is only processed on the server side.
- Users are explicitly given control over permission settings (read, write, delete, edit), allowing the user to decide on how when and how there data is shared with others.
- The way the token generation is implemented, ensures that only the intended recipient can gain access to the stored payload using the microservice only in order to access a file. This supported with providing confidentiality.

3.3 Social Considerations

The system provides a platform for secure collaboration and responsible sharing of data. By integrating a flexible access control system, it can support secure file sharing in production settings where trust, privacy, and integrity are critical.

However, there is a potential social risk if the system is used to share illicit or harmful content. To avoid scenarios like this the following can be applied:

- Server-side logging of events (following GDPR rules).
- Optional file content scanning, for when the microservice is integrated into production.
- A terms of service agreement for users, ensuring ethical use of the system. With actions taken for those who brake the agreement policy.

3.4 Professional Considerations

During development, professional guidelines where followed in order to provide reliability as well as transparency applying the following considerations:

- Ensuring software reliability through automated and manual testing.
- Being transparent about system limitations (e.g. encryption strength may vary slightly between devices due to dynamic resource based key derivation and algorithm selection).

3.5 United Nations Sustainable Development Goals (SDG)

The project aligns with a few United Nations Sustainability and Development Goals [20], in particularly the project aligns with aspects of both Goal 9 and 16:

- **Goal 9 – Industry, Innovation and Infrastructure:** It contributes by exploring scalable, performance-aware encryption and secure digital infrastructure.
- **Goal 16 – Peace, Justice and Strong Institutions:** It supports data integrity, accountability, and privacy which is critical in institutional use cases.

3.6 BCS Code of Conduct

This project adheres to the BCS (British Computer Society) Code of Conduct [5], which outlines the professional standards and ethical responsibilities expected.

For public interest, the system has been developed with a focus on user privacy and data protection. The token based access control implemented ensures that the payload data is encrypted and accessible to only authorised users, assisting in preventing unauthorised access to user data. The expiration time of the applied access and the included permission sets prevent misuse of shared files.

In regards to professional competence and integrity, all the encryption methods and access control schemes used in the system were researched and implemented based on general standards. The technologies used are chosen with an understanding of their limitations, especially with the limitations applied when it comes to the environment, such as platforms with low system resources.

The project also follows the understanding of data privacy. Although the developed system does not handle sensitive data directly, handling of all file content has been followed using ethical means providing the user full autonomous access to there data. Allowing them to remove the data completely off the system at there own discretion and share it in a manner of their own choosing.

This project provides a demonstration for the development of secure systems even on hardware limited platforms. By documenting design decisions and challenges, contribute to the shared understanding and advancement of ethical computing practices.

4 Design Choices

4.1 General Design Choices

The system will include two main microservices and a front-end interface. The two microservices will consist of an ABAC microservice and a Postgres database and storage bucket system. Each microservice will run on its own docker container and communicate with each other to provide access and permissions to the files stored in each individual bucket. The ABAC microservice will be written in Typescript in conjunction with Express.js to provide a Restful-API connection to the front end.

The ABAC MS will follow a modular design technique in order to enhance maintainability and long-term support. By structuring the system into independent components, each function can be modified or extended with minimal impact on others, as long as the provided outputs and inputs remain consistent. The system is divided into two main components: (1) the **System Performance Adjuster**, which dynamically optimises encryption schemes based on resource allocation, and (2) the T**Token Management Components**, which handle token generation, decoding, and ABAC enforcement.

The system will include the following components:

Component	**Details**
adjustSystemPerformance.ts	This component is in charge of deciding on which algorithms and key generation limitations will be applied by using a benchmarking tool on the system resources available.
generateToken.ts	Creates the token based on parameters passed in from the front end UI insuring the token is generated and shared securely
decodeToken.ts	An internal component which simply decodes any provided token within the microservice. With no external use or connection to the front end.
redeemToken.ts	Applies the decodeToken component and updates the table of used tokens accordingly whilst also applying authorisation checks on the token. Returning the decoded payload as an output.
applyAccess.ts	Uses the decoded payload from redeemToken to stream the file data and apply the predefined permissions to the receiving user. Authorisation checks are applied once again.
fetchSharedFiles.ts	This function gathers all files shared with the logged in user and returns them as a short lived signed URL in order for the user to gain access to the file.
fileManagement.ts	This function is in charge of defining the ABAC rule sets that will be applied based on the token's permission field.

Table 4: Microservice Components

4.2 Database & Bucket Storage

The database will include a two table's to track JTI (JWT Identifiers) and their validity duration, avoiding direct storage of the JWT tokens for increased security. As well as include a table which keeps track of currently applied access rights. The table storing the JTI's will also include a boolean column that is a fixed value that can only be set once. This field will be used to tell the system whether the JTI has been used before hence in addition whether the JWT has been used before or not. Supabase [48] will be used for the database setup as it offers a streamlined approach to PostgreSQL with a built-in dashboard for easy management. Additionally, its dedicated JavaScript package enables intuitive and straightforward database interactions, simplifying implementation.

Supabase also offers its own version of S3-compatible bucket storage [49]. This allows for seamless file management with built-in access controls, enabling secure and efficient storage for the project goals. Each user will have a dedicated storage bucket, ensuring that only the owner can access its contents. Access can also be granted to authorised users with specific permissions, allowing for controlled and secure file sharing.

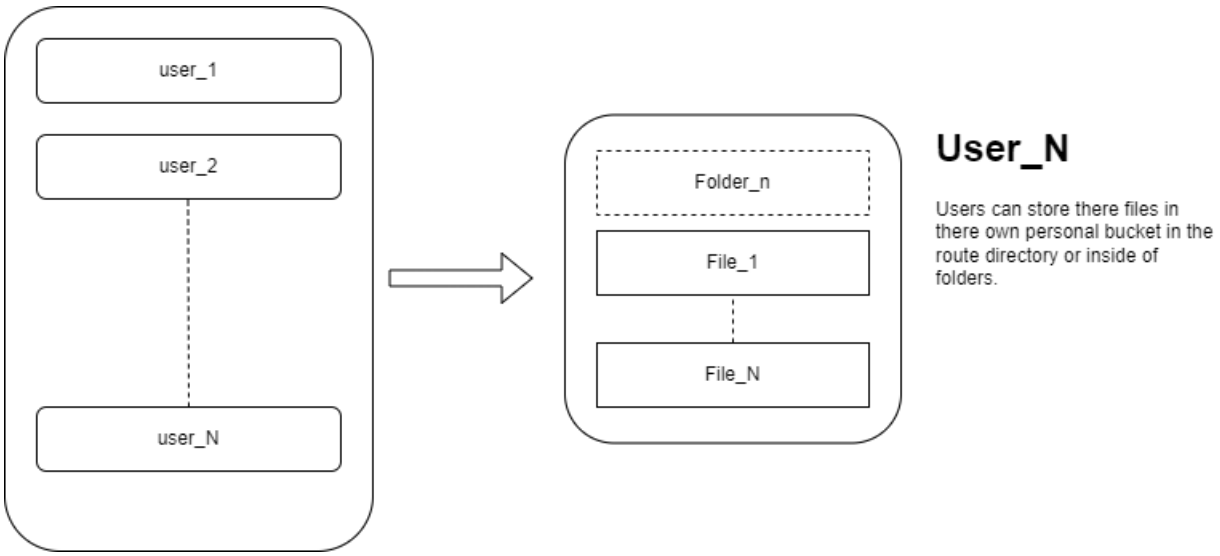

Figure 5: Basic Illustration Of The S3 Storage Layout

4.3 Authentication

In order to provide secure and scalable user authentication, Clerk Auth will be taken advantage of. Clerk was chosen because of the high level of security included and its compliance with regulatory standards such as the Health Insurance Portability and Accountability Act (HIPAA) [21], which is relevant when handling sensitive user data. The service also allows for the configure of custom log-in solutions such as using third-party identity providers using OAuth 2.0. In addition, Clerk provides a seamless front-end integration process with Next.js 15, which accelerates the development process, Allowing the focus to be more on the authorisation platform itself rather than user authentication. Furthermore, Clerk also supports simple two-factor authentication providing an additional level of security and protection to the system.

4.4 Front End

The front end will consist of a simple demo user interface which will comprise mainly of a file management system. The file manager will provide the user with the ability to upload new documents to their own bucket, as well as access to all the files currently stored within the bucket. The system will have a separate section for files shared with the user. This will be a separate page to which the user's own files a stored. A file of choice can be shared with the receiving user via a URL or a QR code. The QR code can be scanned in order to get the same URL.

The interface to grant access to a file will consist of a drop down menu to select the file of choice, listing the files in alphabetic order which will allow you to find the exact file you want to share. Followed by a similar drop down menu for selecting the user you want to share the file with, with their display name being shown. The user can then select the permissions they would like to provide to the receiving user. The token will only be generated if the minimum requirement is set which is the "Read" permission being selected. Write, Edit and Delete permissions can also be applied. This will allow the receiving user to be able to edit the file, delete or download the file based on which permissions are applied. Following these set variable's the user will set an expiration time in the format of **dd/mm/yyyy hh:mm**.

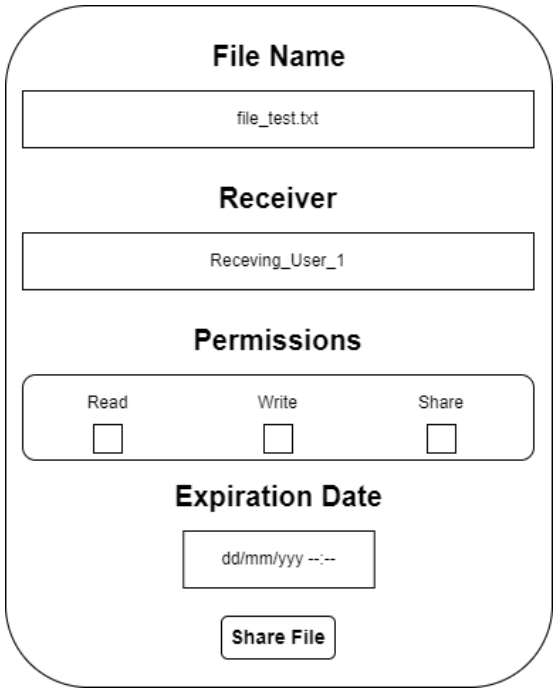

Figure 6: Illustrated Example Of The Access Granting Interface

When the form is submitted a Rest-API call is made to the ABAC Microservice, the service processes the request generating a valid single use JWT token. The token is then returned to the front end with a POST request and the JWT is provided to the user within a modal pop-up as a shareable link and QR code. The form can be accessed on the **/grant-access** page which can only be accessed when a user is logged in. Otherwise, it is hidden.

A logged in user can view their own personal files on the **/my-files** page.

4.5 Token Generation

In order to define the token parameters, the forwarding user directly controls the key parameters of the JWT. The user selects the file to be shared, defines the token's validity duration, designates the recipient user, and assigns corresponding permissions of their choice (e.g., Read, Write, Share). This is accomplished with a frontend user interface as stated above. The user interface can only be accessed by authenticated users and the authenticated user can only share access to files they own.

In designing the JWT token payload for the ABAC system, Fine-grained access control is prioritised while ensuring minimal token size to optimise performance. The payload of the JWT includes the following:

Payload	**Details**
Sender ID	The identifier of the forwarding user (This is also the identifier for the Bucket in which the file is located in)
Receiver ID	The ID of the user who can access the decrypted token
File	The name and route of the file to be shared with the recipient
Permissions	An array of permissions applied to the receiving users (e.x. Read, Write, Share)
JTI	JSON Token Identifier used to make the token unique (e.x. 6966d92c-1f35-4ea5-89d1-43fdc8c31b9d)
Expiration Time	The time frame allowed for the token to be valid and usable in minutes
JWT Algorithm	The algorithm used to sign the JWT based on the system resources and decided by the system performance microservice (e.x. HS256 with A128GCM)

Table 5: JWT Payload

This design aligns with the ideologies of ABAC, providing attributes such as reading and writing that allow ABAC decisions rather than role-based constraints such as RBAC. By embedding these constraints directly into the token, the microservice can enforce policy-based dynamic access rules.

Additionally, the inclusion of file identification and permissions enables stateless authorisation, reducing database lookups for policy checks while maintaining flexibility. However, to avoid large and over-detailed tokens, only essential attributes are included in the payload, with additional policy details referenced externally when needed, such as the checks applied to see if the token has been previously used or if the correct user is trying to redeem the token.

To enhance the security and uniqueness of the tokens issued, A JSON Web Token Identifier (JTI) claim is included in the token payload. The JTI is used as a unique identifier for each token generated, preventing token reuse by checking a database of JTIs. When a token is generated, the JTI will be passed to a database which will store the JTI with a boolean variable to determine whether the token has been used previously or not. If the token has been marked as used, the

record in the database can no longer be updated and will permanently show the JTI as being redeemed, preventing the JWT from being used in further redemption requests.

A secret key will then be generated using a Key Derivation Function (KDF) which will include a passphrase set by the owner of the ABAC microservice along with a salt to add additional randomness. In generating the KDF key, a pre-defined set of iterations will be applied to reduce the ease of dictionary and brute-force attacks being attempted. A set length and digest will also be included to provide a more complex and unique secret key. These parameters are not arbitrarily selected but are instead dynamically determined using the system performance metrics provided by a metric algorithm tested on the system resources. This allows for an additional set of randomness for the secret key.

Once all the payload data has been gathered or defined, the receiving user's ID and chosen file can be encrypted using the CryptoJS library [7] using an AES based encryption scheme. This library was chosen due to its large range of supported AES schemes as well as its support for typescript. Once these fields in the payload are encrypted the JWT token can be signed including the expiration time and JWT algorithm. Following the token being signed and created, the token itself is encrypted using a token encryption function from **JOSE** [12] using an encryption scheme selected from the dynamic algorithm function. After the JWT is encrypted an API call is forwarded onto the front-end which creates a shareable URL and a QR code generated using the **react-qr-code** package [18] so that the person generating the token can share it with the intended individual.

4.6 Token Decoding & Redemption

When a user receives a token, they have the ability to access it via a provided link or by using a corresponding QR code. Once the link is accessed, the token is transmitted to the ABAC microservice using an API POST call, along with the identity number of the currently logged-in user. All computations related to authorisation and access control are performed exclusively within the microservice, which minimises the risk of data breaches by preventing unauthorised modification of the token data (these could be sensitive information such as file details or recipient credentials) since the decoded data is never passed back to the main system. The token is decrypted and verified, following this the encrypted internal variables are decrypted providing the system with the payload data.

The decoded JWT token is used only after verifying that the user trying to decode it has the appropriate rights to access the enclosed information contained within the token. Provided that the user is validated and is authorised to access the token data further, the decoded token is then passed on to the token redemption algorithm in order to apply the permissions to the user, which will allow them to have verified access to the intended file.

During the token redemption process, the user who attempts to gain access to a file will have to provide their token. The next step will be decoding, which will happen by providing a verification of the token and its contents using the decode function. Provided that all checks have passed and the token and the user successfully passed the verification process, the system then checks if the token has previously been redeemed or not. This check-up process is done by referencing the list of JTIs stored in the database. If the JTI has been marked as redeemed the

token is automatically revoked. If the token has previously been redeemed, the decoded data is kept secure in the microservice and a denial error message is forwarded to the user.

If the token has not been used previously, then the decoded token's payload is provided to the system to apply the requested access to the user provided they pass all identity checks and that the token has not yet expired.

4.7 File Access and Applying Permissions

For the requesting user to gain access to the file provided, the system will do the checks stated in the previous section. Once these have all passed and the user has permission to access the token content, the system will access the decoded token payload as a JWT payload object. The current process will separate all the content in the payload into individual variables and once again the system checks if the logged-in user and receiving user id's match. This is done in order to verify one last time if the user has access rights but in addition to also checks if the permission set includes permission to read the file or not. If no read permission is set, access is denied.

Once all this is done, the system will create a signed short life URL in order for the user to load the content of the file locally without the main address and the link of the file being shared. This signed URL is only accessed in the back-end and is streamed to the user's front-end interface. If the signed URL cannot fetch the file, and no POST Request body is retrieved the system throws an error. Given the file has been received, the user is streamed the file content using a node.js stream function.

4.8 Accessing Shared Files

In order to access shared files, a call is made to the database which stores currently valid and used token information. The database returns all records which include the logged in user's user ID providing a list of files they have been granted access to and a list of files they have shared themselves. Files shared to an individual are provided as a list with a generated short life signed URL which is valid for 60 seconds only. Once the 60 seconds expire the link becomes invalid, if the user is still authorised to access the file after this 60 seconds time has passed the link will be refreshed to provide them with renewed access to the file. A list of files shared by the logged in users to others will be provided by showing who they have shared it with and for how long. In addition a revocation button will be available to cancel a user's access before it has been expired. This is done by removing the record from the database early before the user's access time has finished and the record has been removed automatically. Due to the fact that the nature of the JWT that was shared to gain access is a single use token only, the user can not gain access again until they receive a new token.

4.9 Dynamic Algorithm Optimisation

Algorithm selection takes effect on the first token generation. This can be triggered by running a test on the system or when the first user tries to generate a token for the first time. During the algorithm selection, each profile level is tested starting from the least secure and making its way up the stack. Each test case has a hard limit of 3 seconds to be processed. If the algorithm takes longer than 3 seconds to generate a valid token, then the algorithm prior to this attempt is selected. The reason a 3 second upper limit is applied is to ensure that the system does not get overloaded when multiple requests are sent to the system at the same time. During testing, it was determine that 3 seconds provided the best balance between user experience, efficiency, and security. If the processing time takes longer, the user is more likely to think the system has crashed and will re-request the token generation further taxing the microservice.

During testing each algorithm, multiple aspects of the system are taken into account. However, while the processing time is the most important metric, system-specific details such as the number of processor cores and the amount of system memory are also taken into account. The reason these additional metrics are considered is because the system may have had less process than usual running along side the ABAC microservice. This may have effected the result of the dynamic algorithm allocation for the system. If the system takes 3 seconds to run the lowest profile on a system then it will chose the lowest profile available in order to try and provide the best possible security with user experience trade-offs being taken to achieve this.

The profiles are defined by having tested multiple scheme variations on various levels of hardware limitations. The best-case scenario is tested on high end levels of hardware that provide eight or more cores and 8 GB of system memory or more. The lowest profile is tested on a single core system, such as a single board system with 512 MB of system memory. Other levels of profiles are tested on these same systems, as well as platforms that limit the resource allocation to the microservice in order to get a wider idea even if the hardware was not available to natively test the system performance. To mimic these systems, emulation services such as QEMU [17] paired with docker are taken advantage of. Using QEMU the microservice can run in an environment that emulates a range of hardware levels. Additionally, the environment can be further tuned by restricting the amount of cores and system memory docker is allowed to utilise for the microservice container.

5 Implementation & Project Documentation

5.1 Basic Elements

The structure of the implementations consists of 3 core components:

- Frontend
- ABAC Microservice
- PostgreSQL Database & S3 Bucket Solution

API Specification

The system provides a RESTful API that allows the microservice to communicate with the frontend. It is responsible for secure ABAC permissions over shared files. Below is an overview of each API endpoint as well as its HTTP request type:

- **/generateToken** - *POST* - Generates a signed JWT token embedding ABAC permissions including sender/receiver identifiers, file ID, permission set, and an expiry time. The token is dynamically configured based on system resource availability.
- **/decode-token** - *POST* - Validates and decodes a JWT token against the expected receiver ID. This ensures that only the intended recipient can access the token payload.
- **/redeem-token** - *GET* - Allows a user to redeem a token to obtain access to a file shared with them.
- **/shared-files** - *POST* - Retrieves the list of files that have been shared with the current user by providing their user id. This is used to provide the user's Received Files section on the frontend.
- **/files-shared** - *POST* - Retrieves a list of files that the user has shared with others, providing the user with a list of files they have shared, which they can revoke access to early if needed.
- **/remove-access** - *POST* - Revokes access to a previously shared file.
- **/file-management** - *POST* - Manages all ABAC requests. Cross-referencing the current user's permission set and providing access accordingly.

General Overview

The Frontend is structured to provide the end user with a simple and fully functional platform, providing enhanced routing and layout capabilities for development. It includes the following pages:

- **/my-files** – Allows authenticated users to view and upload files. Initially saved to the 'public' directory, however was later replaced with Supabase Storage Buckets (S3 Buckets) for improved file management and security.
- **/my-files/[filename]** - Loads the selected file and is displayed to the user.

- **/my-files/edit** - Allows the user to edit the currently selected file. Providing a user interface to view and modify the content of the file.
- **/grant-access** – Enables users to generate access tokens with a URL and QR code for file sharing. This page interacts directly with the ABAC microservice and database, gathering user details and list of files they can share. In addition, it also calls the ABAC microservice in order to generate a token with the specified metadata.
- **/shared-files** - Users can view the list of files shared with them by other users and also see what files they have shared allowing them to revoke access to a file and user at any time before the access time expires.
- **/reedem-token** - This route is responsible for handling token redemption. It passes through URL parameters, that are the access token and corresponding user ID of the user who shared the file.

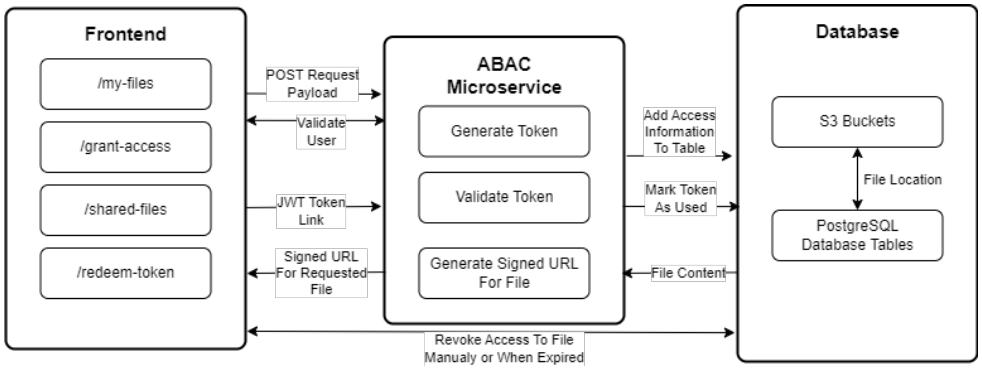

Figure 7: System Overview

Each page communicates with the database and the ABAC Microservice as required. Each user is provided a page that displays all of their files, enabling them to view, share and modify their files. In addition, the user can upload new files through this page using a file upload form. The ABAC Microservice is accessed through RESTful API requests (GET, POST). When the user wants to share a file, the form on the frontend creates a POST request to the microservice. The data selected in the form such as the file to be shared, whom it is being shared, with and the permissions provided to the user, are passed through to the microservice using the POST request. The microservice then runs the **generateToken** script. The script places all of the data into the JWT payload. When the payload is set, a secret key is generated based on the current microservice settings, allowing the key to be generated efficiently and securely. Predefined parameters are provided by the system such as the length, pass phrase, and salt of the secret key. A JTI identifier is also generated at this step and added to the payload for the token. Once all the data is collected and every requirement is generated the JWT is then signed with a system set algorithm and is provided back to the user on the frontend as a URL and scan-able QR code.

When the URL is accessed or the QR code is scanned, an additional request is made to the microservice backend. The microservice decodes the token without providing any information to the end user, and it compares the user id of the currently logged in user with the user id of

the receiving user from the token payload. Whilst the check is being processed, the token's JTI identifier is marked in the database as used, preventing the token from being redeemed again. If the user accessing the token is the intended user, the token payload is decrypted, and the access requested is granted. The file that the user that has gained access to is stored on the database with a timer based on the duration of time the user has been permission to interact with the file. The system uses a real-time communication with the database and the ABAC system to revoke access to the file when this timer expires. The user can view all files they have shared or have been shared with them on the **/shared-files** page and can access the contents of the file from here. All files on the **/shared-files** page use a signed URL with a short life to enable access to the file content, which means that the system regenerates a signed URL for each file every sixty seconds to prevent unauthorised access from being gained. This completes Objective 2, 4 and 9 by providing a fully functional file navigation system as well as a token sharing mechanic with a link and QR code.

5.2 ABAC Microservice

5.2.1 Token Generation

In order to generate a JWT the **jsonwebtoken** package for Node.js is used. This allows a JWT to be provided in order to house the payload for the file access. A function called **generateToken** was designed to perform this task. This is an asynchronous function which allows the code to wait for dependencies to be generated before continuing on to the next line of code that has a pre requisite requirement. This is taken advantage of to generate the performance based encryption settings prior to token generation, initially calling and waiting for the system performance evaluation to process. Once the system performance metrics are provided, the function loads the pass phrase and salt from the environmental variable file **.env** which stores sensitive data that is unique to the current setup of the microservice. This is stored securely and only on the current instance of the service.

```
// System specific variables gathered from the .env.local file
const phrase = process.env.PHRASE!;
const salt = process.env.SALT!;

// System settings applied to the key variables
const it = systemSettings.it;
const len = systemSettings.len;
const digest = systemSettings.digest;

// The JTI being creating using a v4 uuid package
const jti = uuidv4();

const generatedkey = crypto.pbkdf2Sync(phrase, salt, it, len, digest);
```

It loads the system settings generated earlier and generates a unique JTI using **v4** uuid package. These variables are then added into the **crypto.pbkdf2Sync()** function from the Node.js crypto package, as it provides a secure and efficient native solution that generates a secure secret key to be used during the JWT signing process.

After the secret key is generated, the payload data for the JWT is collected from a POST request sent by the frontend file sharing form. The request body below contains information used to provide the rule set and data for the JWT:

```
const { senderId, fileName, receiverID, permissions, expirationTimeInMinutes } = req.body;
```

The receiving user ID, senders user ID and file name are then encrypted using a secondary secret key generated by the system. This secret key provides a slight variation compared to the key used to sign the JWT later down the line. The key uses the same pass phrase and salt as the primary secret key with a different iteration count and a constant **SHA-256** digest with a key length of 32 which does not change based on the system settings. An AES standard encryption scheme is applied to the payload fields, and then they are re-added to the payload replacing the unencrypted versions of the variables.

```
const secondaryKey = crypto.pbkdf2Sync(phrase, salt, it2, 32, 'sha256').toString();

const encryptedFileName = CryptoJS.AES.encrypt(fileName.toString(), secondaryKey.toString()).toString();

const encryptedSenderID = CryptoJS.AES.encrypt(senderId.toString(), secondaryKey.toString()).toString();

const encryptedReceiverID = CryptoJS.AES.encrypt(receiverID.toString(),
↪  secondaryKey.toString()).toString();
```

The payload explicitly specifies the sender and receiver user IDs, the file that will be shared, the permission set applied to the receiving user (read, write, edit, delete) and the token's expiry time. The access control rules are embedded directly within the token, and allows for an ABAC based design, enabling secure and decentralised permission management.

Following this the JWT is signed using the payload data and system settings generated earlier with the expiration date, and the JWT Algorithm also specified. A record of the JTI is then added to the database with a boolean field to state whether the token has been used before. By default the field is set to **FALSE**. Finally, the function returns the URL that can be provided to the receiving user with the token being added in as a URL parameter. The URL is then also used to create a QR code that can be scanned by the receiver for quick and efficient token sharing.

```
const token = jwt.sign(payload, SECRET_KEY, {
  expiresIn: `${expirationTimeInMinutes}m`,
  algorithm: systemSettings.encryptionAlg as jwt.Algorithm,
});
```

Once the JWT token is created and signed an additional step is taken in order to secure the token further, the JWT is encrypted using a system specified encryption algorithm. The algorithms listed by the system are:

- A128GCM
- A128CBC-HS256
- A256CBC-HS512

Using AES the token is encrypted using one of these algorithm. The lowest end of the stack uses an **A128GCM** scheme in order to provide a version of AES that provides proof of authenticity and integrity with the lowest resource requirement during encryption and decryption applying an **AES-128** scheme with **GCM** for MAC based authentication. All profiles other than this apply **CBC** based AES schemes with H-MAC authentication such as **HS256** and **HS512**.

```
const dataEncoder = new TextEncoder();

// Used algorithm: 'dir' for direct encryption since the same service is used for encryption and
↪  decryption
const token = await new CompactEncrypt(dataEncoder.encode(signedJWT)).setProtectedHeader({ alg: 'dir',
↪  enc: systemSettings.alg }).encrypt(generatedkey);
```

Once the JWT has been fully processed and is ready to be used, the JTI related to the JWT is uploaded to the database with the tokens expiration time. This is done to be able to keep track of invalid tokens even if the expiration date has not yet been reached. At the same time a link is generated using the predefined base URL for the host website stored in the environmental variables file. The link includes the encrypted JWT as a URL parameter. Provided no errors have occurred during token generation such as the expiration date being in the past the link is returned to the user using an API call.

```
const expiresAt = new Date(Date.now() + parseInt(expirationTimeInMinutes) * 60000).toISOString();

await supabase.from("tokens").insert([
  { jti, used: false, expires_at: expiresAt },
]);

const link = `${process.env.WEBSITE_LINK}/redeem-token?token=${token}`;


return res.status(200).json({ link });
```

5.2.2 Token Validation & Redemption

In order to validate the JWT the system first requests the encrypted JWT and the logged in user's ID. The decode script is initially run, during this step the encrypted JWT is decrypted using the same secret key used to encrypt it. Following this step the JWT is then verified and the payload is extracted. At this stage the payload includes an encrypted version of the receiver's and sender's user ID as well as the file which they are being granted access to.

```
const { plaintext } = await compactDecrypt(token, new Uint8Array(SECRET_KEY));

const signedJWT = new TextDecoder().decode(plaintext);

const decoded = jwt.verify(signedJWT, SECRET_KEY, {algorithms: [settings.jtwAlg]}) as JwtPayload;

const decryptedSenderId = CryptoJS.AES.decrypt(decoded.senderId,
↪  secondaryKey).toString(CryptoJS.enc.Utf8);
const decryptedFileName = CryptoJS.AES.decrypt(decoded.fileName,
↪  secondaryKey).toString(CryptoJS.enc.Utf8);
const decryptedReceiverId = CryptoJS.AES.decrypt(decoded.receiverId,
↪  secondaryKey).toString(CryptoJS.enc.Utf8);

decoded.senderId = decryptedSenderId;
decoded.fileName = decryptedFileName;
decoded.receiverId = decryptedReceiverId;
```

When the token has been decoded the next step is to verify the users identity. Using Clerk the logged in user's ID can be fetched and is passed through to the microservice. When the microservice receives the ID it will do a comparison of the logged in user's ID and the ID stored within the payload of the decoded token. If the two ID's do not match the system will keep the payload data to its self and return a denied access error to the user attempting to apply the

permission set. If the two ID's match the system will then start the process to apply the access to the logged in user. Firstly whether the user was successfully verified or not the JTI stored in the database is updated to mark that the token has been used. Preventing the token from being reused in the future. If the JTI has been previously redeemed, even if the user passes the validation checks the access to the token is rejected by the system.

```
await supabase.from("tokens").select("used").eq("jti", jti).single();

const { data: finalTokenRecord, error: finalFetchError } = await supabase
    .from("tokens")
    .select("used, expires_at")
    .eq("jti", jti)
    .single();

// Mark the token as used
const { error: updateError } = await supabase
    .from("tokens")
    .update({ used: true })
    .eq("jti", jti);
console.log("Token has been redeemed");
```

Once the token is redeemed the access is applied directly to the user's frontend environment. The token payload and user ID are passed through, once this has happened the system applies an additional check on the payload to confirm authenticity and checks if a list of permissions is included. Provided the payload is valid and the required data is received a signed URL with a limited life time is generated and is streamed to the user.

```
const { data } = await supabase.storage.from(senderId).createSignedUrl(fileName, remainingTime);

const fileResponse = await fetch(data!.signedUrl);

if (!fileResponse.body) {
    return res.status(404).json({ error: 'File not found' });
}

// Convert the web stream to a Node.js stream and pipe it to the response.
const nodeStream = Readable.fromWeb(fileResponse.body as any);
nodeStream.pipe(res);
```

When the JWT is validated successfully and the JTI record is updated. A copy of the applied permissions are saved to the database. The data from the decrypted JWT includes the following:

- File Name
- Sender and Receiver IDs
- Storage Location
- Permission Set
- Expiration Time

This data is stored and securely kept for the length of time the user has access to the file. The access permission is checked constantly using a Real Time call to the database and when the expiration time has been reached the stored data for the specific record is removed automatically.

5.2.3 Access Control Logic

After receiving a request to access a file the system runs its verification checks, firstly verifying the authenticity of the request and if the token is valid.

The system enforces access control by checking that the requester's user ID matches the receiving user's ID within the token and that the specified permissions allow the requested user the **read** permission for file retrieval. Tokens with the incorrect receiving user ID or insufficient permissions are rejected.

Once access can be granted, the system fetches the remaining time until the token expires and generates a secure, time-limited signed URL to access the requested file from the storage bucket. This signed URL is only valid for a maximum of 60 seconds before it is regenerated. If the access time permitted is less than 60 seconds it will last for the specified amount of time only.

```
return await Promise.all(
       data.map(async (file) => {
           // URL is valid for 1min
           const { data: fileData, error: fileError} = await
           ↪  supabase.storage.from(file.bucket_id).createSignedUrl(file.file, 60);

           if (fileError) {
               return console.error(`Error generating signed URL for file ${file.file}:`, fileError);
           }

           const timestamp = file.validity

           return {...file, signed_url: fileData.signedUrl, timestamp};
       })
   );
```

To enforce the expiration of the token, the current timestamp of the system is compared with the expiration field of the stored token payload. If the token has expired, access is then revoked. The time stamps are rounded down to allow for a consistent comparison. If the time stamp of the expiration time is less than the current time then the file is marked as expired and access is removed, and the stored record of the access is removed.

```
   const expiredFiles = data.filter(file => {

       const fileTimestamp = Math.floor(new Date(file.validity).getTime() / 1000);

       return fileTimestamp < Math.floor(Date.now() / 1000);
   });
```

The file is retrieved from storage and streamed directly to the user through a server response from the microservice. This allows for a controlled access environment for content delivery. This is done to ensure that only authenticated and authorised users can access shared content.

5.2.4 Attribute-Based Access Control in File Operations & ABAC Authorisation

Attribute-Based Access Control

All file operations exposed by the microservice such as downloading, deleting, and updating the file are protected by a central Restful API route that evaluates requests based on the ABAC policies stated within the user's JWT token. If a permission is not encoded within the JWT then the API route ignores the unspecified permissions and only applies the permissions present in the token at the time of redemption.

Each operation is assigned a specific permission requirement:

- **download** - downloadFile (**async**) - A signed URL is generated with a lifetime of 60 seconds, which is used to retrieve the file as a BLOB which is a file object type that can be provided as text and binary data and allows the user to download the chosen file.
- **delete** - deleteFile (**async**) - The file is deleted from the bucket it is stored in and any record of it being shared is also removed.
- **update** - updateFile (**async**) - Using a BLOB (Binary Large Object) of the file to pass updated data to the bucket of type **text/plain**.

Each permission has its own function that is run when executed. The functions are only streamed to the user provided they are granted access. Locking away the functionality of permissions they are not granted. For the permissions that have been granted the user is provided with a button and dedicated pages for each function where necessary, and these UI elements are loaded from the server side of the front end only when allowed. The API call from the microservice passes the rule sets to the frontend on page generation and are securely generated accordingly.

Authorisation

In order to insure these permissions are enforced, each ABAC function requires validation checks from both the microservice and the data base in order to stream the data to the user.

On pages such as **/shared-files** the logged in user is fetched directly from the authentication service and is crossed referenced with the shared files table in the database. Only if the logged in user has a record matching the file and bucket they are trying to access. Given this is included in the table the user is then provided the file in order to view or modify it. If they are unsuccessfully authenticated, a 401 is thrown and it is stated to them that they are unauthorised to access the contents of that file.

Similarly with functions such as **update**, the user has to be verified through the authenticated and only then are they allowed to access the file within the file editor.

5.2.5 Encryption Schemes

A key feature of the ABAC microservice is its ability to dynamically adjust encryption schemes based on available system resources. This ensures that the system can maintain acceptable performance levels even on resource-bound devices while still offering a reasonable level of security.

The encryption profile is chosen during the first run time of the system. When the first JWT generation is executed, the system resources are inspected and the encryption profile is chosen accordingly. The number of CPU cores, memory allocation and overall system load is considered during the choice of the encryption profile.

System Profile	JWT Alg	KDF It	Key Len	Digest	AES Scheme	AES It
High	HS512	800,000	64 bytes	SHA-512	A256CBC-HS512	300,000
Medium	HS384	800,000	32 bytes	SHA-384	A128CBC-HS256	50,000
Low	HS256	800,000	16 bytes	SHA-256	A128GCM	10,000
Extra Low	HS256	400,000	16 bytes	SHA-256	A128GCM	1,000

Table 6: Encryption Profile Configuration Based on System Resources

Each profile reflects a trade-off between cryptographic strength and processing efficiency. The best case configuration uses HS512 for the JWT Algorithm, and an A256CBC-HS512 scheme is used to encrypted the JWT using AES. It provides high resistance to cryptographic attacks and is suitable for systems with reasonably high levels of system resources.

Fallback profiles, in particular the extra lightweight profile that uses HS128 for the JWT Algorithm and A128GCM to encrypt the JWT, are used for systems with extremely restricted resources. Although these profiles provide a lower level of security and are easier to crack, they enable the system to function responsively without crashes or timeout errors when a functioning system is required.

The Key Derivation Function (KDF) uses the **crypto.pbkdf2Sync** function of the Node.js crypto library [6] with dynamically tuned parameters. A digest algorithm (**SHA-512, SHA-384, or SHA-256**) is applied based on the set profile and the iteration count (**KDF It**) directly affects the strength and computational cost of generating the secret key used in the JWT.

5.2.6 System Optimisation and Adjustments

In order to dynamically select a encryption profile, the system performs a benchmark on all the available encryption profiles. Starting from the lowest level of security and working its way through the stack. Having set a 3 second upper limit, which is based on system testing allows the system to maintain a high enough level of security without effecting user responsiveness or overwhelming the system in the future when multiple requests are processed at the same time. Preventing the system from failing to generate tokens and or from being out of service due to too many requests overloading the platform.

The benchmark measures the average time taken to generate a JWT token in all encryption profiles. Each profile was evaluated multiple times whilst being decided upon in order to get the best values possible and the results were averaged to ensure accuracy. When the system

is deciding on the best algorithm to use it runs an instance of every profile starting from the weakest and making its way up to the strongest algorithm, however when the system comes across a profile that takes longer than 3 seconds to process, it fails that profile test and keeps the current best performing scheme as the most optimal profile to use on the system. During this process a dummy payload is used in order to mimic the use of a real file and permission set to ensure token generation times were realistic. It included sender and receiver IDs, file name, and ABAC permission set along with a validity time and JTI.‘

```
  const generatedkey = crypto.pbkdf2Sync(phrase, salt, scheme.it, scheme.len, scheme.digest);
  const secondaryKey = crypto.pbkdf2Sync(phrase, salt, scheme.it2, 32, 'sha256').toString();

  const encryptedFileName = CryptoJS.AES.encrypt("TestFile.txt", secondaryKey).toString();
  const encryptedSenderID = CryptoJS.AES.encrypt("test_userId_76552gs783gd", secondaryKey).toString();
  const encryptedReceiverID = CryptoJS.AES.encrypt("test_receiverId_12345", secondaryKey).toString();

  const payload = {
    senderId: encryptedSenderID,
    receiverId: encryptedReceiverID,
    fileName: encryptedFileName,
    permissions: ['read'],
    jti: uuidv4(),
  };

const signedJWT = jwt.sign(payload, generatedkey, {
    expiresIn: `50m`,
    algorithm: scheme.jtwAlg as jwt.Algorithm,
  });

await new CompactEncrypt(new TextEncoder().encode(signedJWT)).setProtectedHeader({ alg: 'dir', enc:
↪  scheme.alg }).encrypt(generatedkey);
```

The payload above is the dummy data used in each profile during profile testing. It includes basic data such as the user ID's of the sender and receiver, file name and permission set. In addition, it includes the unique JTI generated using **uuidv4()** [22]. Taking advantage of the CryptoJS [7] the file name and the sender's user ID are encrypted using a AES encryption scheme similar to the main system. Once the payload is set the token is then signed and encrypted again using the CompactEncrypt function from JOSE [12]. Two secret keys are generated, one for signing and encrypting the token and a secondary secret key based on different parameters used during the process of encrypting internal variables.

5.3 Bucket Implementation

In order to provide a secure and isolated storage solution for each individual user on the platform. Supabase's S3 Bucket solution was used [49]. Using a bucket bases storage solution provides security as well as efficiency to file management. It uses an object based storage solution enabling easy file upload and download as well as simplified access control schemes for the included files. For the project each user is provided with their own individual bucket, when a user goes to upload a file. Initially a check is made in order to see if the user already has an existing bucket, and if not it creates a new storage object for the logged in user using their credentials.

```
const { data: existingBuckets, error: bucketError } = await supabase.storage.listBuckets();

if (bucketError) { throw bucketError;}

const bucketExists = existingBuckets.some(bucket => bucket.name === userId);

if (!bucketExists) {
    const { error: createBucketError } = await supabase.storage.createBucket(userId, {
         public: false,
});

    if (createBucketError) {
        throw createBucketError;
    }
}
```

A form is provided to the user to upload files to their storage bucket, the functionality requires the user to be authenticated before being able to view the form.

The storage buckets are accessed in order to provide third-party users with access to an individual's file. Assuming a user has been granted access and all checks have been applied by the microservice, the bucket of the file's owner is accessed using the user ID of the owner from the payload. This is directly retrieved from when the requesting user after they redeemed the JWT. A list of all files the user has been granted to is displayed to them with a short life signed URL. The user can access the content of the file from the bucket using this signed URL.

An additional list of files shared by the currently logged-in user is also provided, allowing them to see what files they have shared. The data for this is directly fetched from the database records of shared files that match the current logged-in user's ID. The logged-in user can revoke access to the files they have shared. These files also are delivered using a signed URL from the bucket object with a 60 second life span before they are regenerated in order to prevent an unauthorised user from being able to gain unauthorised access to the files. A real-time check was also implemented in order to track and delete files access permissions as they expire, when an access right expires the record is removed from the database but the file stored in the bucket remains uneffected, allowing the owner of the file to re-share the access rights if they wish. With the bucket implementation set **Objective 5** is completed successfully.

6 Testing

6.1 Testing Scenarios

A series of testing scenarios were undertaken. The main area which was tested was the token generation time, which is crucial for finding a good balance in security and performance. By measuring generation times across different system loads and hardware environments as well as different encryption levels, it was possible to fine-tune the encryption parameters dynamically in order to provide a best case for a large variety of hardware configurations. In particular, the system was tested on a high end system such as my own system which ran the tests on an i7-12700H which provided 14 cores to be used, paired with 64 GB of system memory. In addition, the system was emulated on Raspberry Pi [40] platforms using QEMU [17]. The hardware restrictions were altered to allow for the microservice to be able to lower the available core count as well as memory to get an idea of how that would effect the system.

In addition to performance, the security of the generated tokens was evaluated. Tests were conducted to verify that:

- The tokens payload could not be accessed without the correct secret key.
- The key derivation function produced sufficiently unique secrets to prevent them from being brute forced.
- The encrypted values within the JWT were secure and could not be read before the system decrypted them, even when intercepted during transmission.
- The encryption of the JWT was also tested to ensure that even if the secret key was discovered, the AES encryption applied to it prevented it from being deciphered.

Negative test cases were performed using expired and invalid tokens to confirm the resilience of the token validation function and that unauthorised access attempts were denied and rejected.

6.2 Testing Environments

In order to test the API routes, which included tests such as concurrency testing. Postman [15] was taken advantage of using its local client. This allowed me to test API payloads, concurrent users over a specified amount of time, as well as to test the system when invalid fields were provided.

- **Individual Test Time:** 5 minutes
- **Number Of Concurrent Users:** 1, 5, 10 and 20
- **API Requests Tested:** *POST* /generate-token
- **Metrics Looked At:**
  - Average response time
  - Error rate
  - System resource usage (CPU, memory)

  - Throughput (requests/sec)

- **Purpose:** The test helped to observe the system performance under varying levels of stress, particularly its response to multiple file share requests. This helped to see how well the encryption and ABAC logic remained within acceptable levels of performance under realistic usage spikes.

Invalid Data Testing

- **Objective:** To check that the microservice provides appropriate responses when API requests include incomplete, or unauthorised requests.
- **Postman Tests:** Using predefined requests with invalid payload and JWT input.
- **Test Scenarios:**
  - Missing required fields in payload (e.g., no receiver_id, file missing).
  - Invalid JWT token or expired token usage.
  - Incorrect payload data, such as undefined permission sets.
  - Invalid permission when the user with no **download** permissions tries to download a shared file.
  - Sharing files with no ABAC permissions provided in particular when no **read** permissions are present.
- **Expected Behaviour:**
  - Error message and request rejection when a payload is sent with missing or incorrect data.
  - Denied access for when invalid or expired tokens are provided.
  - Denial of service when incorrect access attempts are attempted based on the user's provided access rights.
- **Findings Summary:** All tested endpoints successfully rejected invalid input as expected, returning appropriate status codes and error messages. The ABAC microservice handled invalid requests successfully in the test scenarios, preventing any unintended file access.

6.3 Invalid Data Testing

To ensure the robustness of the system, a series of tests were conducted to validate how the API handles invalid or malformed token requests. These included scenarios such as expired tokens, altered payloads, and missing parameters.

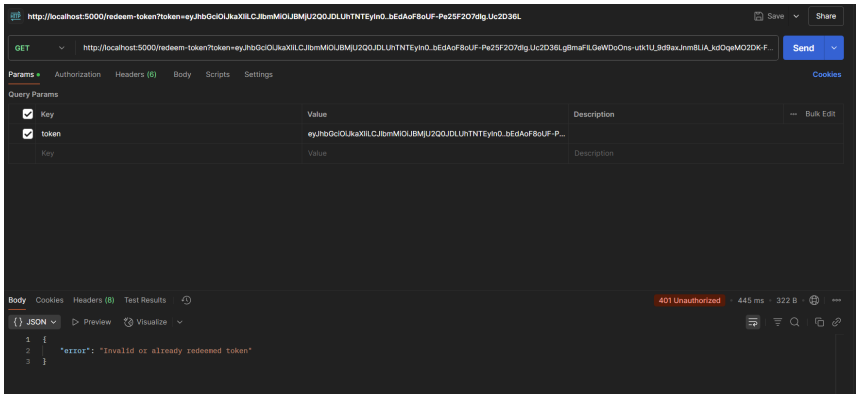

Figure 8: Expired or Invalid Token Test

Figure 8 demonstrates the response of the service when a user attempts to redeem an expired or invalid JWT. The ABAC microservice correctly rejects the request and returns an error message:

```
{
  "error": "Invalid or Already redeemed Token"
}
```

This confirms that the API applies validation checks before processing and applying redemption requests, preventing access attempts based on incomplete or modified data.

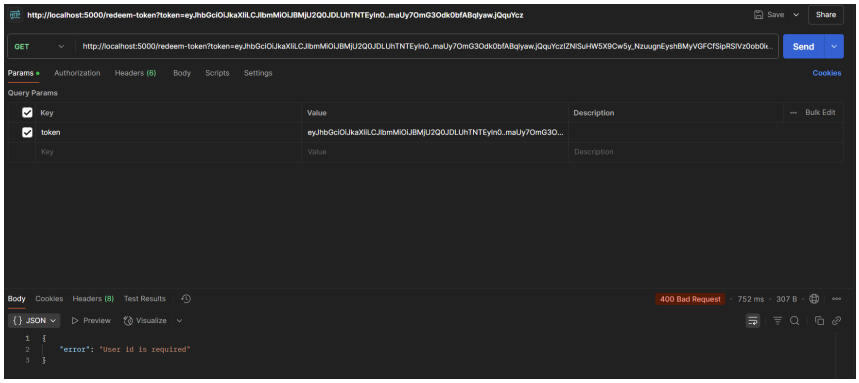

Figure 9: Missing Valid User ID on Redemption

Figure 9 shows the system response when a token redemption is attempted without providing a valid user ID from clerk auth. In this scenario, the user is not logged into the service and because of this does not provide the required `userId` parameter. As a result, the system returns the following error message:

```
{
    "error": "User id is required"
}
```

This confirms that the microservice has built in identity validation during the token redemption process, ensuring that only authenticated users can access resources if permitted.

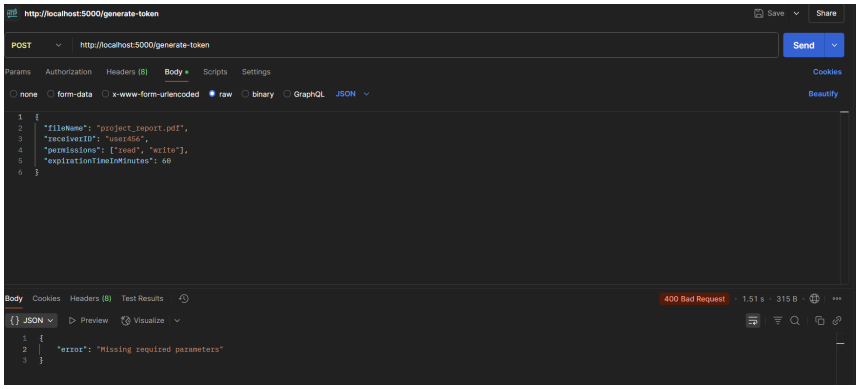

Figure 10: Missing Payload Parameter

Figure 10 demonstrates a test in which a token generation request is established with a missing field within the payload. When the system detects a missing field, the system automatically rejects the request and prevents the token generation from taking place, and returns an error message as follows:

```
{
    "error": "Missing required parameters"
}
```

6.4 Algorithm Performance Metrics

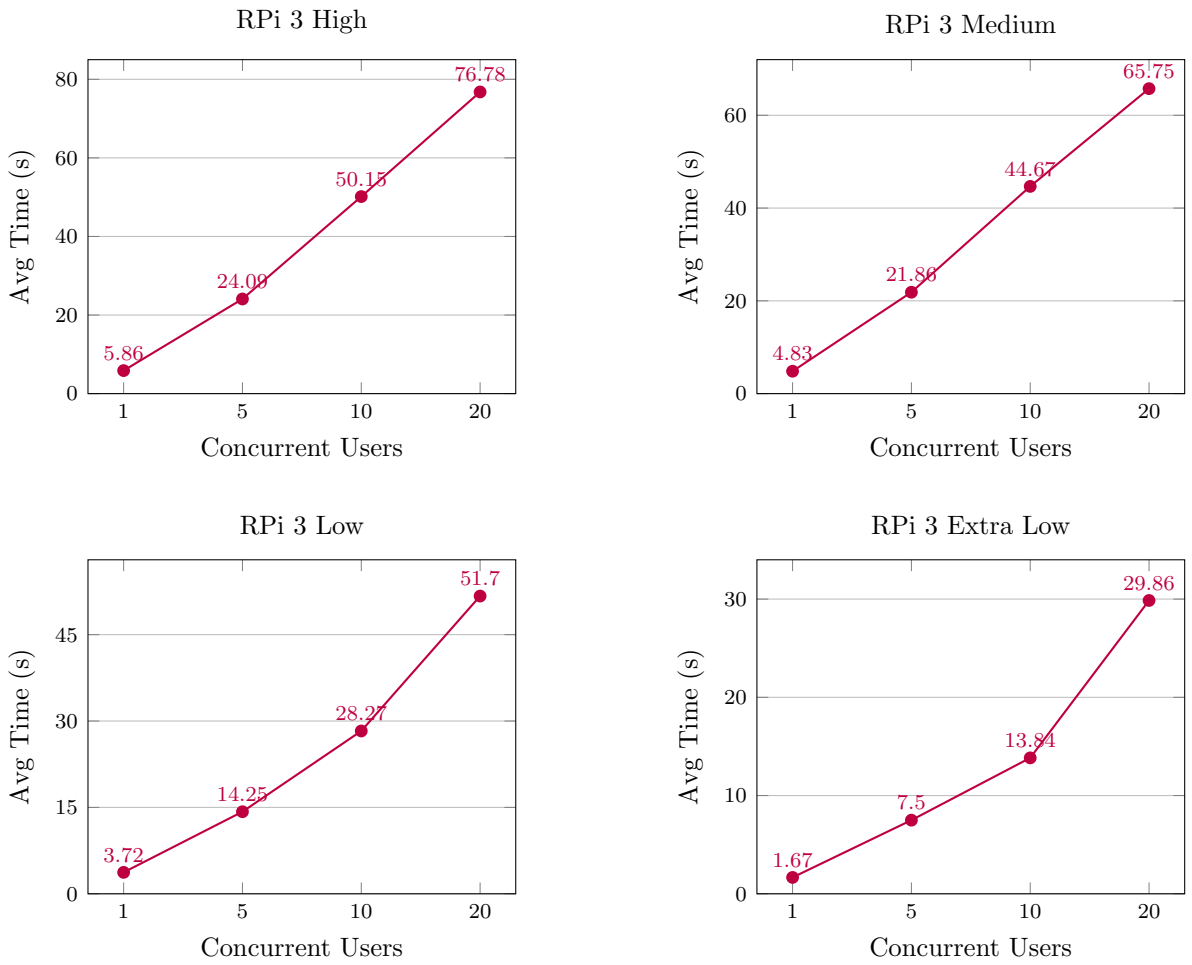

Figure 11: Token Generation Time in Seconds Based on Number of Concurrent Users for Various Encryption Profiles on a RPi 3 Emulation (4 core, 512MB Memory)

During testing on a Raspberry Pi 3 (RPi) emulated via QEMU [17] with the various profiles Table 6. I observed a significant performance degradation with the **High** encryption profile in particular as concurrent users increased. Initially, the increase in time taken to process requests appeared acceptable. However, once the number of users exceeded five, the system began to fail under load. Showing this profile is unsuitable for the Raspberry Pi 3. The **Medium** profile performed better in comparison to the **High** profile processing twenty concurrent requests faster than twenty individual ones. However, its scalability was still limited, as the curve increased drastically after ten concurrent users. These results suggest that while stronger encryption provided additional security, it caused scalability issues that resulted in the **High** and **Medium** profiles being unusable in this situation. In comparison, the **Low** profile provided a more acceptable curve increase in token generation time. While latency rose with more users, it remained within more acceptable limits. Showing that this profile provides an ideal balance for this hardware set. The Extra Low profile offers the best scalability and responsiveness. Under 20 concurrent users, average token generation time remained under 30 seconds. Showing that this profile is

ideal for performance, however, it provides major security trade-offs. For the Raspberry Pi 3 the ideal profiles would be the **Low** as well as **Extra Low** profiles. The dynamic algorithm system chose the **Extra Low** profile in this situation. Providing headroom for the system in case of performance fluctuations due to the three second maximum tolerance of the system.

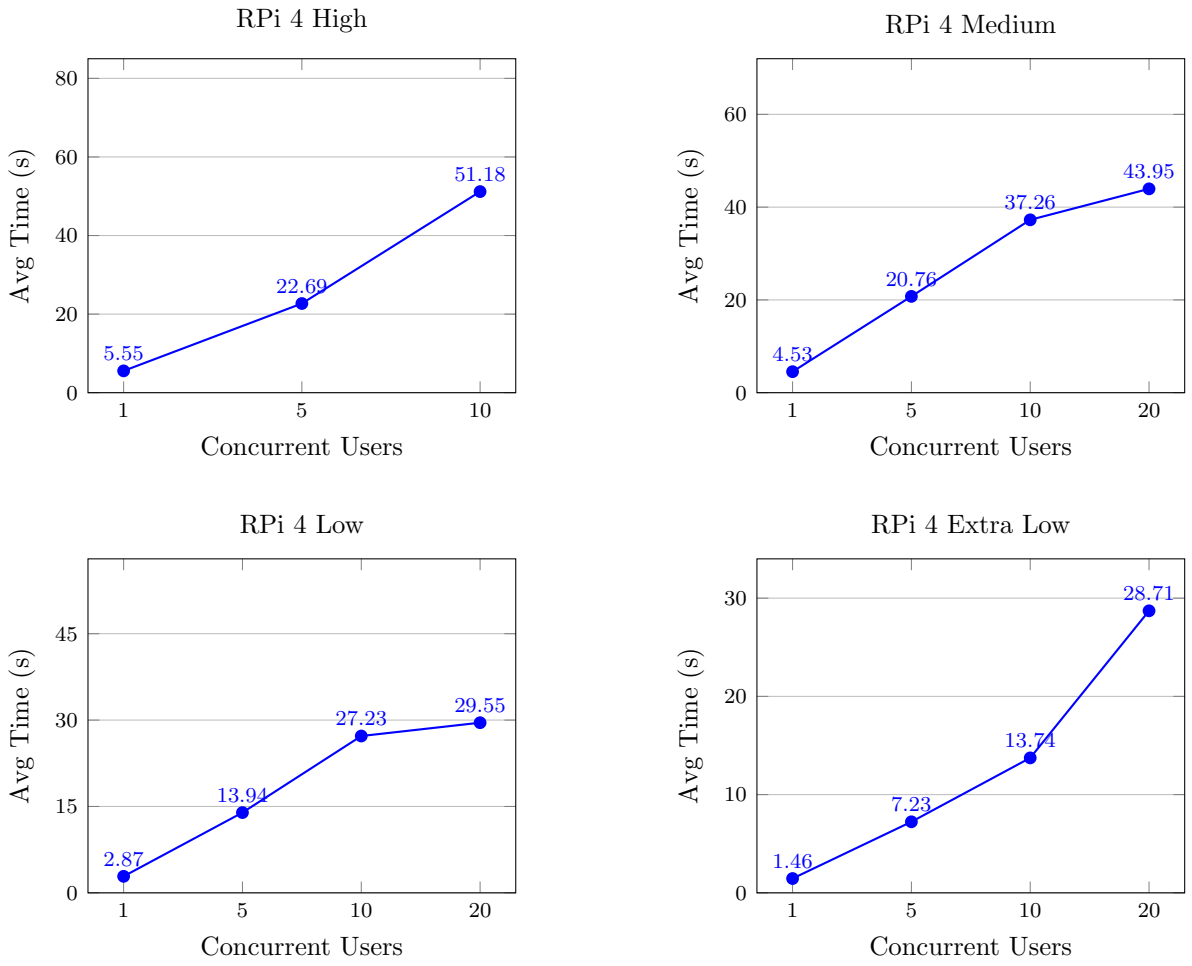

Figure 12: Token Generation Time in Seconds Based on Number of Concurrent Users for Various Encryption Profiles on a RPi 4 Emulation (4 core, 4GB Memory)

Figure 12 shows the result for testing the platform on a RPi 4 emulation with 4GB of system memory. The system was seen to perform better overall compared to the RPi 3 emulation, which only had 1GB of system memory available. When a single user is using the system at a time, it was demonstrated that the performance difference appears to be minimal between the two platforms. However, as the concurrent users increased, the performance difference began to show and the benefit of the additional system memory started to be taken advantage of. The **High** profile provided a reasonable increase compared to each test case, however, the system was unable to handle 20 users with this profile. On the medium setting the computation time remains linear until it reaches the final test case of 20 users in which it takes drastically less time to process the requests compared to a 10 users request being run twice.

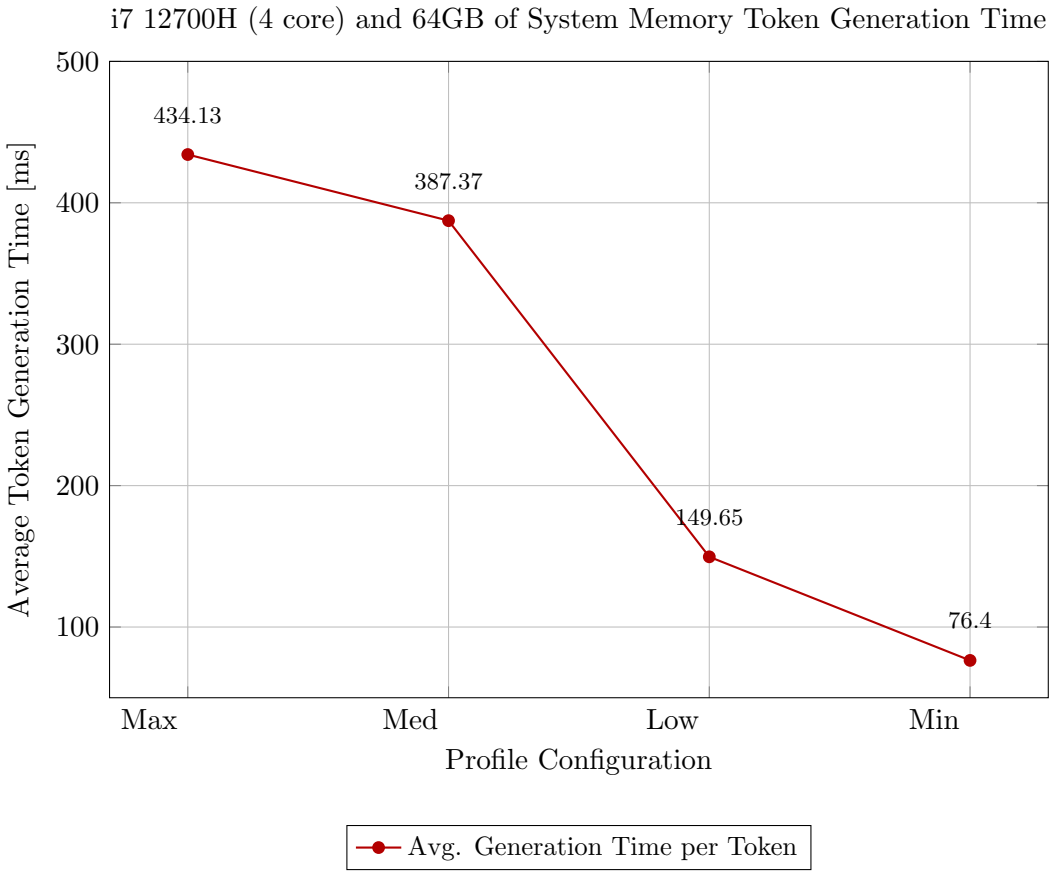

Figure 13: High Performing System Test

The graph in Figure 13 shows the average token generation time across the four profiles available on a high end system. The system used a i7-12700H processor, which was limited to 4 cores. As the configuration increased from **Extra Low** to the **High** profile, the average generation time drastically increased from 76.4 ms for the **Extra Low** profile configuration to 434.13 ms for the **High** profile configuration. This showed that the system was capable of handling the highest profile provided that the longest time taken to generate a JWT was well below the 3 second threshold. The system automatically selected the **High** profile when the benchmark was run to determine the profile choice of the system. The system was able to take full advantage of the provided profiles and could even be used to process even more computationally expensive profiles if available.

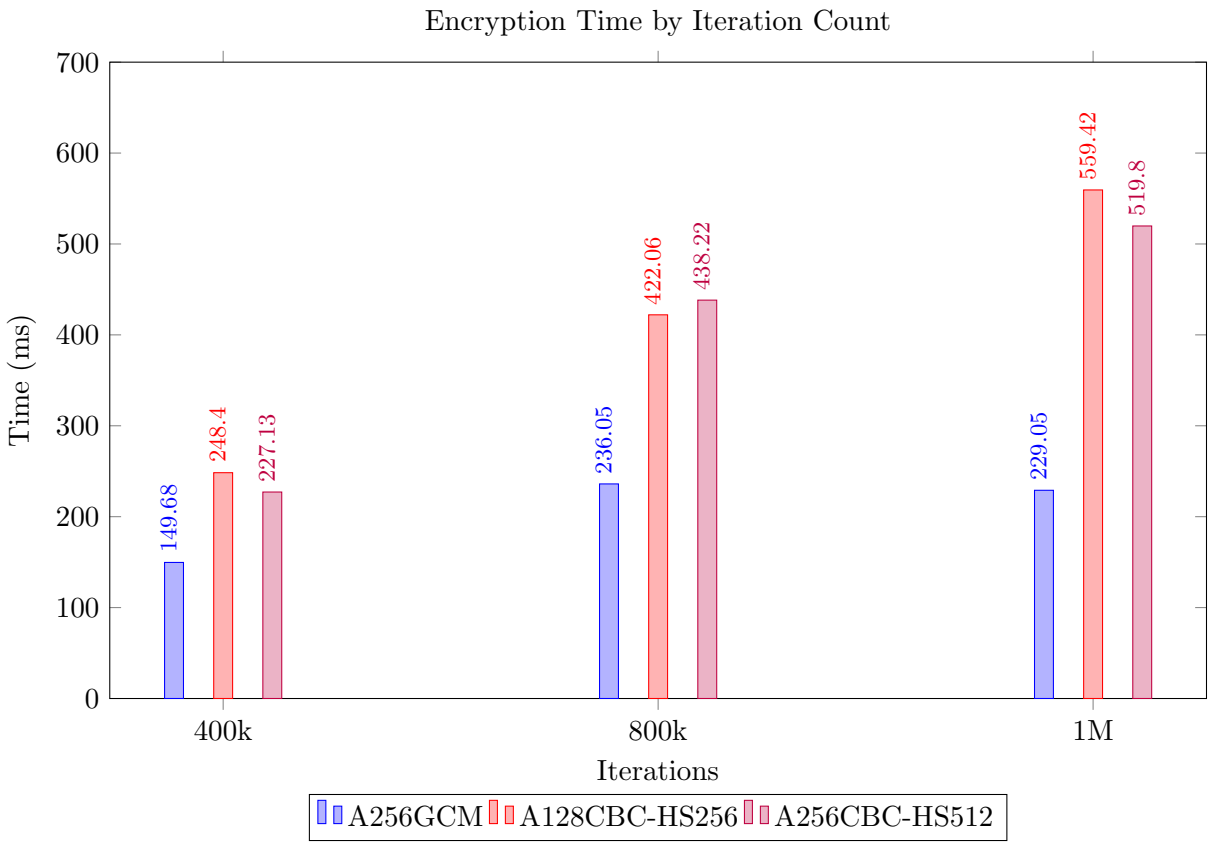

Figure 14: JWT Encryption Schemes Performance (iteration fixed at 100k for internal variable encrypter)

To evaluate the performance of the different encryption schemes used to generate the JWT tokens, a controlled benchmark was completed on three profile configurations: **A256GCM**, **A128CBC-HS256**, and **A256CBC-HS512**. These were tested with an increasing PBKDF2 iteration count: 400k, 800k, and 1M, with the internal variable encryption iteration count fixed at 100k.

The goal of the test was to determine how much of an effect the iteration count would have on the token generation time. From the test, it can be seen that the A256GCM scheme consistently outperformed the other schemes in terms of speed, achieving the lowest encryption times across all iteration counts. At **1M** iterations, the token generation was completed in 229 ms compared to 559 ms (A128CBC-HS256) and 519 ms (A256CBC-HS512). As the iteration count increased, the encryption time significantly increased for the CBC profiles, with **A128CBC-HS256** showing the most drastic increase. This shows that GCM is a more efficient profile, but also scales less dramatically in high-security scenarios that demand higher iteration counts for key derivation. This shows that the GCM based profile is more suited for low powered systems as it can provide higher iteration counts when applying the PBKDF2 key allowing for the security to be increased. However on systems with higher system performance CBC based models would provide the overall greater security. In concluding this Objectives 7 and 8 are fulfilled with the information gathered from the testing applied to the predefined encryption profiles available. Similar results to these where viewed with different hardware configurations as well, providing a rough idea on

the general functionality of these different profiles.

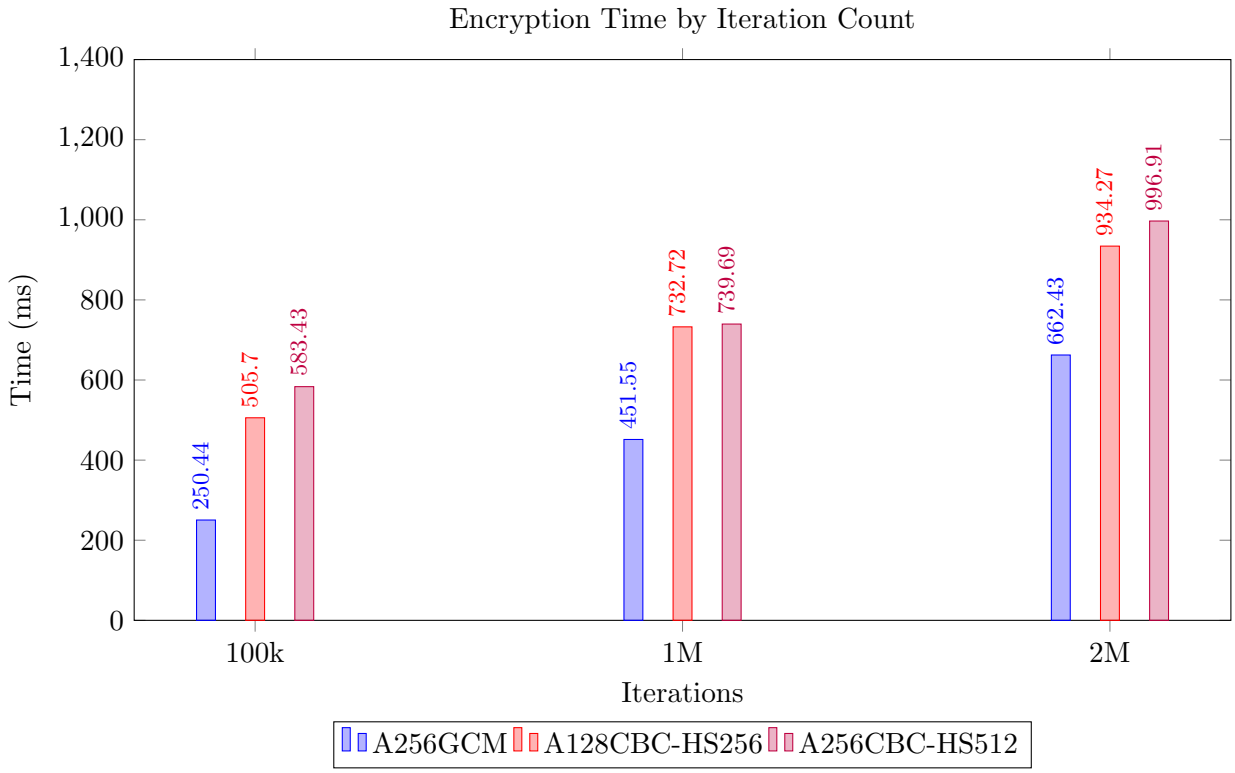

Figure 15: Internal Variable Encryption Schemes Performance (iteration fixed at 800k for JWT encrypter)

Figure 15 presents a performance comparison of three different profiles with an AES-based encryption schemes used to secure internal variables in the JWT token. The iteration count in the PBKDF2 key derivation function varied across 100k, 1M, and 2M for these internal payload variables. This iteration count directly affects the performance costs required to generate the secret key used to encrypt these internal variables, influencing the overall token generation time. As shown, the profile which supports **A256GCM** consistently outperforms the other schemes in terms of speed similar to the previous tests run, making it suitable for low end hardware no matter the iteration count on the secret key. The cryptographic strength increases with iteration count, and more secure schemes like **A256CBC-HS512** and **A128CBC-HS256** come with higher processing costs. Notably, **A256CBC-HS512** provides the largest performance degradation at **2M** iterations, highlighting the trade-off between computational cost and efficiency. This analysis shows once again the importance of dynamically selecting encryption profiles based on system hardware, in which lower-end systems may prefer **GCM** based profiles with lower iteration counts, and systems with higher system resources can afford the computational overhead of **CBC** based profiles for enhanced security. In conclusion the test showed that the impact of the AES encryption on the internal variables has a similar impact as the secret key used to sign and encrypt the token which includes these encrypted variables within the payload. However whilst the **GCM** based profile provided the best performance whilst sacrificing security. Both **CCM** profiles performed similarly showing that a drastic effect can not be seen when increasing the internal variable iteration count in these profiles. This test provided the most consistent results.

6.5 Functionality Testing

To begin the functionality testing phase, a set of test files are preloaded to the Supabase S3 bucket of the logged in account. Figure 16 shows how the user initiates the file sharing process by selecting a file stored in their own storage bucket and specifies the recipient user. Using the Grant Access form, the user is able to define which permissions the recipient user will be granted using the ABAC system. The set payload data is then forward securely and are handled by the backend ABAC microservice. This forms the basis for confirming the successful functioning of the access control and token generation components.

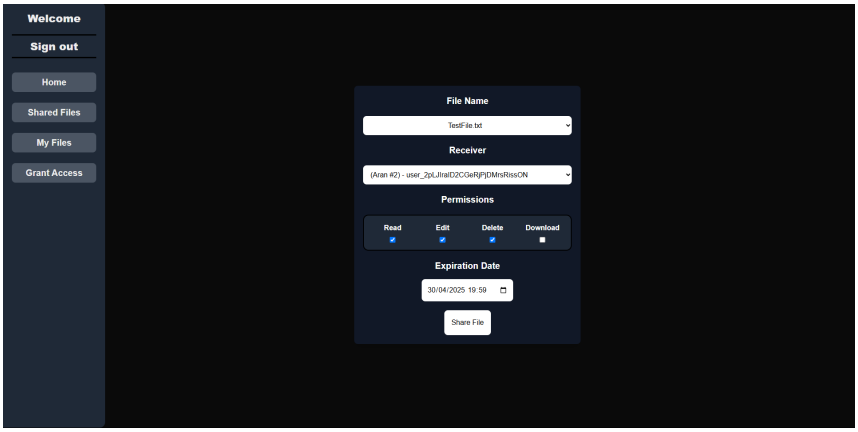

Figure 16: Grant Access Page

Figure 16 displays the Grant Access UI used to start the process of securely sharing files between users. This user interface allows the user to select a file stored in Supabase S3, choose a recipient by their user ID and display name, and assign permissions (read, write, delete, edit) based on an ABAC approach. Once submitted, the form sends the data to the ABAC microservice, which generates an encrypted JWT link containing the access policy. The token is encrypted using a dynamically selected algorithm based on current system resources to balance performance and security. The resulting access token is then provided in the form of a link as well as a QR code as seen in Figure 17 for the receiving user to scan, which allows the receiver to securely retrieve the file within the permitted scope and apply it to their account.

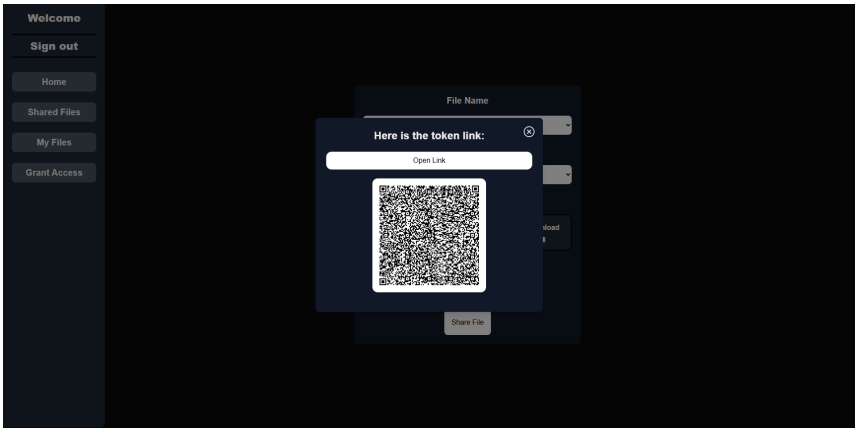

Figure 17: QR Code Access Example

A user can view the files they have uploaded on the my-files page. This page can also be used to upload files to the logged in user's storage bucket as seen in Figure 18. The user can also use this page to modify, download, or remove files. In addition the user can directly begin the sharing process for a specific file by clicking the share button on the side of the file. This will direct them to the form shown in Figure 16 with the pre-selected file of choice. This completes objective 2.

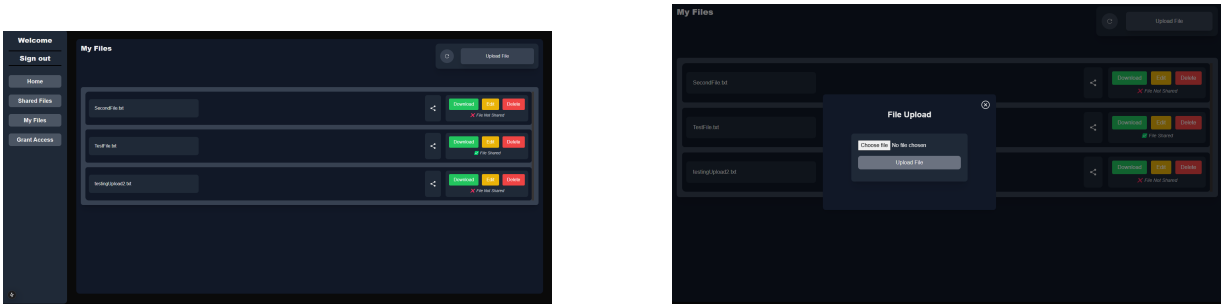

Figure 18: View User Files and Upload Files

As seen in Figure 19 files shared with and by a user can be viewed on the **shared-files** page, this page provides two sections. The first section provides files they have been granted access to and have redeemed the token for, here they can interact with the file they have been provided and the permission sets they have been granted. If the user has been granted "editing" permission, they will have access to the editing menu of the file, and if they have access to download the file they will be presented with a download button.

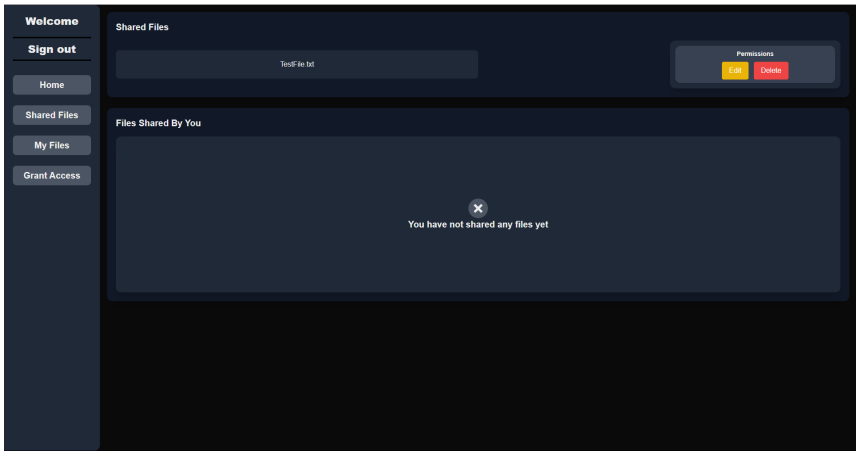

Figure 19: Shared-files Page Receiver

In the second section of the page, as seen in Figure 20 the user can view files they have shared with others. A button is presented on the side of the file in order to allow the user to revoke access prior to the expiration of the access permissions.

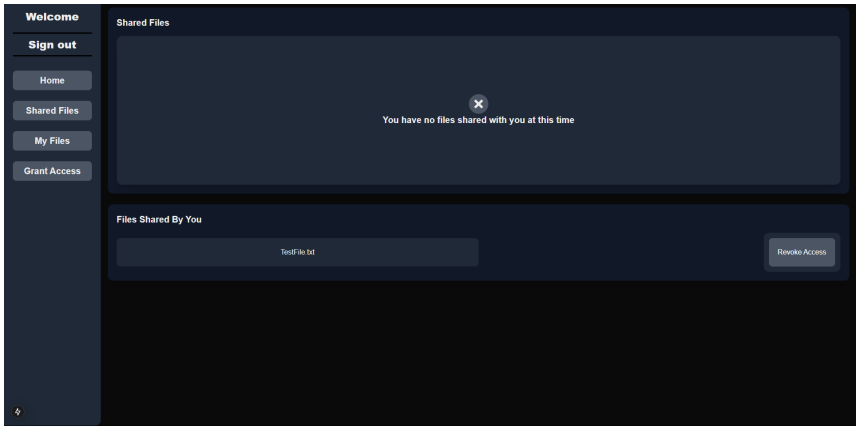

Figure 20: Shared-files Page Sender

When checking a token that has already been used, a check is made to the list of stored JTI's and if the token has been used previously it will be rejected and the decoded token will not be provided to the user. An error message as seen in Figure 21 will be returned and the redemption attempt is ended.

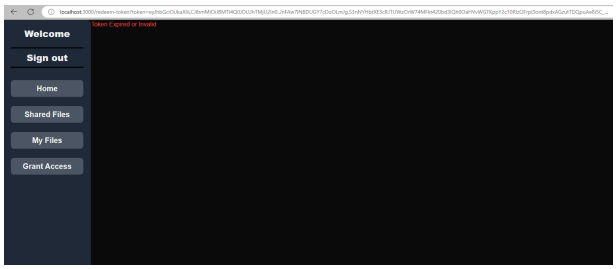

Figure 21: Example of an Already Redeemed Token

In order to edit a files content, a check is applied to ensure that the user has the appropriate edit permissions or that they own the file. If the user is permitted to edit the content, they can click the edit button on the side of the file. The user will then be provided with a space in which they can edit the file as seen in Figure 22.

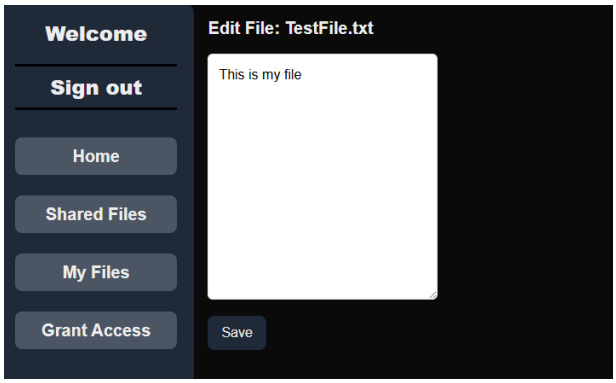

Figure 22: File Editing Example

When a token is redeemed successfully, the user is prompted with the file content, as well as a button to navigate them to the list of all files shared with them as seen in Figure 23. Once redirected, they can find all the ABAC permissions applied and can access them accordingly.

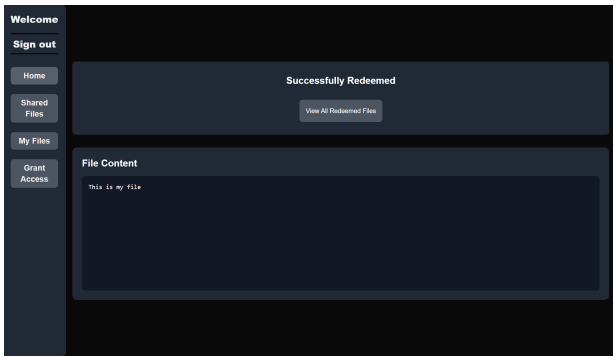

Figure 23: Successful Token Redemption

The first time that a token generation is initiated, the dynamic profile selection is run. The system tests each profile until it reaches a profile that does not get processed in the allocated time frame. As seen in Figure 24 the system ran each profile and in this scenario selected the highest profile marked as **High**.

```
2025-05-10 19:20:54 abac-microservice-1  | ABAC Micro-service running on port 5000
2025-05-10 19:21:02 abac-microservice-1  | Extra Low setting completed in 0.10s
2025-05-10 19:21:03 abac-microservice-1  | Low setting completed in 0.17s
2025-05-10 19:21:03 abac-microservice-1  | Medium setting completed in 0.42s
2025-05-10 19:21:04 abac-microservice-1  | High setting completed in 0.51s
2025-05-10 19:21:04 abac-microservice-1  | Selected scheme: System is optimal, using hig
h-security encryption.
2025-05-10 19:21:04 abac-microservice-1  | System settings saved to systemConfig.json
```

Figure 24: Dynamic Algorithm Processing Terminal

7 Final Evaluation & Reflections

7.1 General Evaluation

The project has overall met its core objectives of enabling secure, controlled file sharing through a MSA based ABAC system. The integration of JWT's, dynamic encryption strategies, and Supabase S3 storage has allowed for a fully scalable implementation. Functional requirements such as file upload, permission-based sharing, QR code/token link generation, and access validation were completed and tested. The system was successfully tested on a low end platform as well as a high end system to get a range of metrics with available resources. Most of the secondary additional features were added. A front-end user interface was developed to meet the needs of the system providing an easy to navigate platform. Although the system demonstrates good performance and validity, there are areas where further refinement would improve reliability.

Nevertheless, the outcome of the project is a fully functional system that demonstrates how modern access control and encryption techniques can be integrated into efficient and low resource heavy products.

7.2 Issues Faced During Development

Issues With Supabase CLI

During development, I encountered a drastic issue with the Supabase CLI tool [47], in particularly regarding the S3 bucket system which is essential for enabling file management in my project. While attempting to interact with buckets locally using the Supabase CLI, I found that file storage operations (e.g., uploading or retrieving files) were consistently failing without clear error messages. The error messages provided stated that Row Level Security was being voided when trying to access or create a bucket. I attempted to apply rule sets to the bucket system configuration, however had no success. Exploring further for a few weeks, despite repeated troubleshooting, including environment setup validation, permissions checks, and examining logs, I was unable to resolve the issue locally.

This limitation forced me to pivot to using the online hosted Supabase [48] web interface instead, where the bucket functionality worked as expected. Although this allowed me to continue progress with file storage and retrieval, it meant that I could not fully test ABAC permissions and JWT-based file access in a local environment to the same extent as I would otherwise have been able to. This experience highlighted a limitation in the platform and its documentation for local development with Supabase's S3 Bucket solution features.

In future iterations, a potential solution would be to isolate and mock the storage layer for local testing or to raise an issue with Supabase's maintainers to improve CLIs support for bucket storage usage.

Issues During Testing

During testing, limitations were faced during the usage of the Postman software when doing concurrency tests on the API routes. Postman provides a maximum of 25 concurrency requests a month, which severely limited the ability to fully test the possibilities of the platform in this regard. I was able to get a reasonable amount of these tests completed each month during the

final stages of the development cycle. However a replacement software that provided the same amount of insight to the tests could not be found in time. Due to this Postman was not replaced.

7.3 Possible Future Additions

Additional Encryption Schemes

One direction for future additions involves adding additional encryption schemes such as ChaCha20 as an alternative to the AES-based encryption currently in use. ChaCha20 is recognised for its speed and security, particularly on devices with limited hardware acceleration for AES, such as IoT or embedded systems.

Although I initially considered integrating ChaCha20 into the microservice, this would not be easily achieved with the time I had due to the current secret key generation method using **crypto.pbkdf2Sync**, which outputs a string rather than a raw byte buffer directly. Since ChaCha20 requires a 32-byte binary key and a specific nonce structure, the string output from the KDF function was incompatible, without additional conversion requirements adding more steps to the systems increasing execution time, but in addition could have risked problems when it came to intended security guarantees or performance. Due to my discovery of ChaCha20 late in the project, I hope to be able to implement it in the future.

Dynamic Scheme Compatibility and Backward Decryption

Currently, the encryption scheme is selected based on system resources with an initial setup taking place, but this is static after the initial benchmark run with the first token generation. Future iterations will feature a fully dynamic encryption strategy where the system will be able to:

- Adaptively select encryption algorithms based on more live performance metrics.
- Automatically detect and apply the correct decryption mechanism at runtime. Allowing for any encryption profile to be processed regardless of the current global scheme setting.

This would ensure that all previously encrypted profiles or tokens remain decryptable, even if the default encryption mode changes over time. This design guarantees both forward adaptability and backward compatibility, critical for long-term system resilience and usability.

7.4 Limitations and Unsuccessful Features

7.4.1 Limitations

Limited Testing Hardware Availability

One major limitation of the project was having limited hardware configurations to test the platform on. Using QEMU emulated environments could be used to mimic hardware stacks such as a range of Raspberry Pi platforms. This allowed for the project to be tested to an acceptable level for low end hardware.

Time Restraints

The amount of time available to complete this project was a major factor in the final product. The core feature set was included as well as improved on. However if more time was available more testing could have been conducted as well as additional features could have been implemented to improve user experience as well as security.

7.4.2 Unsuccessful Features

Mobile Support

Initially at the start of the project, one of the objectives was to provide a mobile version of the system. However, due to time restraints and compatibility issues I was unable to accomplish this goal. The main problem faced in relation to this was being able to find a node package that compiles a Next.js 15 web application into a mobile executable such as an APK file. Having found a few possible packages, an attempt was made to implement this. However several unresolvable issues where faced that prevent me from implementing this objective.

Switching from Next.js 15 to React Native was considered since it is based on the React framework similarly as Next.js, however due to time constraints this was not possible as having a mobile port was one of the final objectives on the list and was left to the end. The web application does provide functionality when accessed on a mobile device through a web browser directly but native support was not successful.

Offline Functionality

The functionality of being able to generate and decode tokens offline was also considered. The project could be migrated to a platform such as Electron in order to provide a local instance of the service. Users could share files cached on the machine using a similar link/qr code method. Link generation could be accomplished using an algorithm such as a Double Ratchet algorithm. This would allow the users to verify each other without the need of making a call to the microservice directly. Using forward secrecy the pre-defined keys can be trusted by both parties. The receiving user can apply the permission to their local instance of the software and when they're connect to the internet they will have access to the shared content. This could also be applied to a mobile version of the software.

Further Sharing

One of the ABAC permissions to be provided aimed to allow a user who has had a file shared with them, to be able to forward the file further to an additional user if they had the permission granted. The maximum time the additional user would have access to the file would not exceed the original pre-defined time frame provided when the owner shared the file. The additional user would only be granted **read** and **edit** permissions to prevent them from downloading the file without the owner's consent.

An additional share panel was to be provided which would apply these secondary permission sets where permitted. When the file is forwarded, the owner would be able to view a log that displays these additional shares.

7.5 Final Thoughts

Finally, this project has allowed me to significantly expand my understanding of access control systems, particularly those based on attribute-based access control. It provided me the opportunity to explore the role of encryption schemes in ensuring data confidentiality and integrity. Throughout the development process I examined how various factors can effect encryption schemes such as computational resources, algorithmic complexity, and key derivation methods. I was able to acquire a strong knowledge on how they influence the reliability and performance of various types of encryption schemes. This hands-on experience not only helped me gain advanced theoretical knowledge but also gave me real world use cases in which I had to examine the trade-offs between security and efficiency in real world situations. I finish this project with a stronger knowledge base for both the technical and conceptual sides required to build secure, performance aware access control platforms. As well as possibilities to further expand the system to provide faster, more secure and more user friendly ways to accomplish this goal.

References

[1] About cloud storage buckets. https://cloud.google.com/storage/docs/buckets.

[2] Apptainer - Portable, Reproducible Containers — apptainer.org. https://apptainer.org/.

[3] auth0.com. https://auth0.com/.

[4] Authorization — keycloak.org. https://www.keycloak.org/docs/latest/authorization_services/index.html.

[5] BCS Code of Conduct for members - Ethics for IT professionals — BCS — bcs.org. https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct.

[6] Crypto (pkdf2sync) — Node.js v24.0.0 Documentation — nodejs.org. https://nodejs.org/api/crypto.html#cryptopbkdf2syncpassword-salt-iterations-keylen-digest.

[7] CryptoJS — CryptoJS — cryptojs.gitbook.io. https://cryptojs.gitbook.io/docs.

[8] Docker — docker.com. https://www.docker.com/.

[9] Express - Node.js web application framework — expressjs.com. https://expressjs.com/.

[10] Information on RFC 2898 &xBB; RFC Editor — rfc-editor.org. https://www.rfc-editor.org/info/rfc2898.

[11] Information on RFC 8018 &xBB; RFC Editor — rfc-editor.org. https://www.rfc-editor.org/info/rfc8018.

[12] Javascript Object Signing and Encryption (JOSE) documentation — jose.readthedocs.io. https://jose.readthedocs.io/en/latest/.

[13] Meeting data residency requirements on aws - aws prescriptive guidance. https://docs.aws.amazon.com/prescriptive-guidance/latest/strategy-aws-semicon-workloads/meeting-data-residency-requirements.html.

[14] Next.js by Vercel - The React Framework — nextjs.org. https://nextjs.org/.

[15] Postman: The World's Leading API Platform — Sign Up for Free — postman.com. https://www.postman.com/.

[16] Public key authentication - security, automatic log-in, no passwords. https://www.ssh.com/academy/ssh/public-key-authentication.

[17] QEMU — qemu.org. https://www.qemu.org/.

[18] react-qr-code — npmjs.com. https://www.npmjs.com/package/react-qr-code.

[19] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (United Kingdom General Data Protection Regulation)(Text with EEA relevance) — legislation.gov.uk. https://www.legislation.gov.uk/eur/2016/679/contents.

[20] THE 17 GOALS — Sustainable Development — sdgs.un.org. https://sdgs.un.org/goals.

[21] User Authentication — clerk.com. https://clerk.com/user-authentication.

[22] uuid — npmjs.com. https://www.npmjs.com/package/uuid.

[23] Ieee standard for low-rate wireless networks amendment 3: Advanced encryption standard (aes)-256 encryption and security extensions. *IEEE Std 802.15.4y-2021 (Amendment to IEEE Std 802.15.4-2020 as amended by IEEE Std 802.15.4z-2020 and IEEE Std 802.15.4w-2020)*, pages 1–23, 2021.

[24] How to securely transfer files with presigned urls — aws security blog. https://aws.amazon.com/blogs/security/how-to-securely-transfer-files-with-presigned-urls/, 06 2024.

[25] Antony Adshead. Storage technology explained: What is s3 and what is it good for? https://www.computerweekly.com/feature/Storage-technology-explained-What-is-S3-and-what-is-it-good-for, 2024.

[26] auth0.com. Jwt.io - json web tokens libraries. https://jwt.io/libraries.

[27] AWS. Cloud object storage — store retrieve data anywhere — amazon simple storage service. https://aws.amazon.com/s3/, 2024.

[28] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. https://doi.org/10.1145/77648.77649, February 1990.

[29] A. Bánáti, E. Kail, K. Karóczkai, and M. Kozlovszky. Authentication and authorization orchestrator for microservice-based software architectures. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018.

[30] Milica Dancuk. How to use public key authentication with ssh. https://phoenixnap.com/kb/ssh-with-key, 08 2021.

[31] Morris Dworkin. NIST Special Publication (SP) 800-38C, Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality — csrc.nist.gov. https://csrc.nist.gov/pubs/sp/800/38/c/upd1/final.

[32] Morris Dworkin. NIST Special Publication (SP) 800-38D, Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC — csrc.nist.gov. https://csrc.nist.gov/pubs/sp/800/38/d/final.

[33] Yanfang Fan, Zhen Han, Jiqiang Liu, and Yong Zhao. A mandatory access control model with enhanced flexibility. In *2009 International Conference on Multimedia Information Networking and Security*, volume 1, 2009.

[34] Zhenhao He, Liji Wu, and Xiangmin Zhang. High-speed pipeline design for hmac of sha-256 with masking scheme. In *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, pages 174–178, 2018.

[35] Syed Zulkarnain Syed Idrus, Estelle Cherrier, Christophe Rosenberger, and Jean-Jacques Schwartzmann. A review on authentication methods. https://hal.science/hal-00912435/, 03 2013.

[36] M. Jones, J. Bradley, and N. Sakimura. Rfc 7519: Json web token (jwt), 2015.

[37] Moin A. Khorajiya and Gardas Naresh Kumar. A security based architecture using kerberos and pgp. In *Proceedings of the International Conference on Advances in Information Communication Technology & Computing*, AICTC '16, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Barry Leiba. Oauth web authorization protocol. https://ieeexplore-ieee-org.surrey.idm.oclc.org/document/6123701, 2012.

[39] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. https://www.sciencedirect.com/science/article/abs/pii/S0950584920301993, 10 2020.

[40] Raspberry Pi Ltd. Raspberry Pi — raspberrypi.com. https://www.raspberrypi.com/.

[41] Djebari Nabil, Hachem Slimani, Hassina Nacer, Djamil Aissani, and Kada Beghdad Bey. Abac conceptual graph model for composite web services. In *2018 IEEE 5th International Congress on Information Science and Technology (CiSt)*, 2018.

[42] Chris Richardson. Microservices.io. https://microservices.io/patterns/microservices.html, 2017.

[43] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf, 08 2020.

[44] Bharath S, Nishanth Kumar Pathi, Shinu Abhi, and Rashmi Agarwal. Accessflex: Flexible attribute based access control scheme for sharing access privileges in cloud storage. In *2024 International Conference on Electrical, Computer and Energy Technologies (ICECET*, pages 1–6, 2024.

[45] Sarita and Sunil Sebastian. Transform monolith into microservices using docker. https://ieeexplore-ieee-org.surrey.idm.oclc.org/document/8463820, 2017.

[46] Prajakta Solapurkar. Building secure healthcare services using oauth 2.0 and json web token in iot cloud scenario. In *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, 2016.

[47] Supabase. Supabase CLI — Supabase Docs — supabase.com. https://supabase.com/docs/guides/local-development/cli/getting-started?queryGroups=platform&platform=linux&queryGroups=access-method&access-method=kong.

[48] Supabase. Supabase Docs — supabase.com. https://supabase.com/docs.

[49] Supabase. Storage buckets — supabase docs. https://supabase.com/docs/guides/storage/buckets/fundamentals, 10 2024.

[50] Supabase. JWTs|SupabaseDocs. https://supabase.com/docs/guides/auth/jwts, 12 2024.

[51] Johannes Thones. Microservices. https://ieeexplore.ieee.org/abstract/document/7030212, 01 2015.

[52] Bayalarm Webmaster. How does an access control system work? https://www.bayalarm.com/blog/how-does-an-access-control-system-work/, 11 2020.