Salle Helevä

IMPLEMENTING CLIENT-SIDE FILE ENCRYPTION FOR AN ENTERPRISE DOCUMENT MANAGEMENT PLATFORM

Master of Science Thesis
Faculty of Information Technology and Communications
Examiners: Associate Professor Antonis Michalas
Doctoral Researcher Eugene Frimpong
March 2023

ABSTRACT

Salle Helevä: Implementing Client-Side File Encryption for an Enterprise Document Management Platform
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
March 2023

M-Files is a document management platform used by enterprise customers. Customers may wish to use M-Files for sensitive documents, the confidentiality of which cannot be trusted with third parties. To this end, a system should be implemented that enables a customer to use M-Files for managing such documents, without requiring trust in the security capabilities of M-Files. This thesis examines how client-side file encryption can be implemented for M-Files. This thesis proposes the M-Files Confidential Document System (MFCDS), a client-side file encryption system. A customer of M-Files can use the MFCDS to create confidential documents, that are encrypted on the client side with keys owned by the customer. The system is integrated as part of the web client of M-Files, using browser-based technology.

An implementation plan for the MFCDS system is presented. Hybrid encryption is used to enable users to share access to encrypted files using public key cryptography. More efficient symmetric cryptography is used for encrypting files. User keys are stored in a remote key management system, owned by the customer. The key management system is accessed via a web API, that implements a simple protocol for key management. The protocol enables envelope encryption and public key infrastructure with user keys.

The proposed implementation plan is followed to its completion, and a proof of concept is implemented. The protocol of the key management API is defined and the API is implemented as a cloud application on the Azure cloud computing platform. The client-side implementation entails changes to the web client of M-Files. The built-in browser-based cryptography module Web Crypto is used for cryptographic algorithms on the client side. A simple user interface is implemented to demonstrate the system in practice.

The efficiency of the implementation is evaluated with performance tests. It is found that the implementation provides good performance for files of a moderately large size. The performance was also found to scale well when the system is used to share encrypted files with hundreds of users.

Keywords: client-side encryption, applied cryptography, web-development, information security, key-management

TIIVISTELMÄ

Salle Helevä: Client-puolen salauksen toteuttaminen yritysdokumentinhallinta-alustalle
Diplomityö
Tampereen yliopisto
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Maaliskuu 2023

M-Files on yritysasiakkaille tarkoitettu dokumentinhallinta-alusta. Asiakkailla voi olla tarve käyttää M-Filesia arkaluontoisille dokumenteille, joiden luottamuksellisuuden turvaamisessa ei voi luottaa kolmansiin osapuoliin. Tähän tarkoitukseen olisi implementoitava järjestelmä, joka mahdollistaa asiakkaan käyttämään M-Filesia tällaisten dokumenttien hallintaan ilman, että asiakkaan on luotettava M-Filesin tarjoamaan tietoturvaan. Tässä diplomityössä tarkastellaan, miten client-puolen tiedostosalaus voidaan implementoida osaksi M-Filesia. Diplomityössä esitetään MFCDS (eng. M-Files Confidential Document System), client-puolen tiedostosalausjärjestelmä. MFCDS:iä käyttämällä M-Filesin asiakas voi luoda omistamillaan avaimillaan salattuja, luottamuksellisia dokumentteja client-puolella. Järjestelmä integroidaan osaksi M-Filesin web-sovellusta hyödyntäen selainpohjaista teknologiaa.

Tässä diplomityössä esitetään implementaatiosuunnitelma MFCDS:lle. Hybridisalauksella käyttäjät voivat jakaa salattuja tiedostoja keskenään julkisen avaimen menetelmää hyödyntäen. Tehokkaampaa symmetristä salausta käytetään tiedostojen salaamiseen. Käyttäjien avaimet säilytetään erillisessä asiakkaan omistamassa avaintenhallintajärjestelmässä. Avaintenhallintajärjestelmään tarjotaan pääsy web-rajapinnan kautta, joka implementoi yksinkertaisen avaintenhallintaprotokollan. Protokolla mahdollistaa envelope-salauksen ja julkisen avaimen infrastruktuurin käyttäjien avaimilla.

Esitetty implementaatiosuunnitelma viedään loppuun toteuttamalla konseptin oikeaksi toteava implementaatio. Avaintenhallintaan käytettävän web-rajapinnan protokolla määritellään ja toteutetaan pilvisovelluksena Azure-pilvipalvelualustalla. Client-puolen implementaatiossa toteutetaan muutoksia M-Filesin web-sovellukseen. Selaimen tarjoama Web Crypto -kryptografiamoduuli toteuttaa vaaditut kryptografiset algoritmit client-puolella. Järjestelmän käytännön demonstrointia varten toteutetaan yksinkertainen käyttöliittymä.

Implementaation aikatehokkuutta mitataan suorituskykytesteillä. Tulokset paljastavat, että implementoitu järjestelmä tarjoaa hyvän suoritusnopeuden kohtuullisen isokokoisille tiedostoille. Suoritusnopeuden huomataan myös skaalautuvan hyvin, kun järjestelmää käytetään salattujen tiedostojen jakamiseen sadoille käyttäjille.

Avainsanat: client-puolen salaus, sovellettu kryptografia, web-kehitys, tietoturva, avaintenhallinta

PREFACE

Writing this thesis was a great challenge for me. The process was as difficult as it was rewarding. Not only did I learn about the fascinating subject of cryptography, but also about writing and thinking. I wish to thank my supervisors Antonis Michalas and Eugene Frimpong from Tampere University, and my company supervisor Esa Kettunen. Their feedback was invaluable, and their guidance throughout the process kept me on a clear track. I also wish to thank my employer, M-Files, for the opportunity to write my thesis about such an interesting topic. I owe thanks to all the colleagues, with whom I had great discussions about the topic as I was writing. Their genuine enthusiasm gave me the motivation to really apply myself. I must also thank my friends and my family. Without the continuous support of all the important people in my life, this undertaking would not have been possible.

Tampere, 7th March 2023

Salle Helevä

CONTENTS

GLOSSARY OF ABBREVIATIONS

AAD	Azure Active Directory
AES	Advanced Encryption Standard
AES-GCM	AES with Galois/Counter Mode
AKV	Azure Key Vault
API	Application Programming Interface
CA	Certificate Authority
CSP	Cloud Service Provider
DEK	Data Encryption Key
ECC	Elliptic-curve Cryptography
ECDH	Elliptic-curve Diffie–Hellman
HSM	Hardware Security Module
JWK	JSON Web Key
JWT	JSON Web Token
KEK	Key Encryption Key
KMS	Key Management System
MFCDS	M-Files Confidential Document System
MFS	M-Files Server
MFW	M-Files Web
NIST	National Institute of Standards and Technology
OIDC	OpenID Connect
PBKDF	Password-based Key Derivation Function
PKCE	Proof Key for Code Exchange
PKI	Public Key Infrastructure
RPC	Remote Procedure Call
RSA	Rivest–Shamir–Adleman algorithm
SaaS	Software as a Service
SPA	Single Page Application

UI	User Interface
UUID	Universally Unique Identifier

1. INTRODUCTION

Digitalization has brought an ever-increasing amount of company data to the cloud. Businesses leverage cloud platforms for storing different types of data, including confidential documents. Since cloud computing distributes functionality to potentially multiple third parties, the cloud service consumer forfeits control of their data, and a level of trust is required [22, p. 5][77, p. 67][45, p. 18]. It is not always clear what level of security the consumer is provided with [22, p. 5-6]. The threat of a security breach on the cloud provider's end and legal regulations are incentives against trusting third parties with the most sensitive data of a company [82, p. 295, pp. 70–72]. These issues can be mitigated by adopting client-side encryption, which refers to the practice of encrypting data with a user-owned key before it leaves the user's device [78, p. 124]. With client-side encryption, the untrusted cloud can be used for storing even confidential information. Client-side encryption gives the end user control and responsibility over access to their data, allowing cloud services to be used without requiring trust in the provider's guarantees for the confidentiality of data at rest. [71, pp. 126–127] This motivation has driven researchers to propose various frameworks [70][56][32].

For protecting the confidentiality of an individual user's data, client-side encryption appears to be an ideal solution, giving the user complete control. However, there are challenges in finding an implementation strategy to meet the expectations of enterprise customers. The customer may wish to control their users' access to company data, which in the case of client-side encryption implies administrative control over key management. Thus, the customer may wish to use their own identity and access management system or a trusted third party one, to control access to encryption keys. Furthermore, key management must follow best practices for security, allowing the customer to, for example, audit user activity and manage key life cycles [13, pp. 43–44, 29–30]. Additionally, as opposed to personal use, with a userbase of company employees, it would be desirable to share access to the same encrypted files between a group of users, posing yet another implementation challenge.

This thesis investigates how client-side file encryption may be implemented for a document management platform used by enterprise customers. The customer is enabled to centrally manage the keys of their users. The users are enabled to share access to encrypted files, allowing collaboration while working with confidential documents. The privacy perspective in this thesis is not on the level of the individual user. As users are often the employees of the customer, the privacy of their personal files is not a focus. Privacy is to be provided on the level of the whole customer enterprise, implying protection for confidentiality from third parties. The goal is to build a client-side encryption system, that allows the customer to use M-files for files of a sensitive nature, without requiring trust in M-Files as a provider of confidentiality.

1.1 Motivation

M-Files Corporation commissioned this thesis with the goal of evaluating the viability of client-side file encryption functionality as part of M-Files, a content services platform specializing in document management. M-Files protects data-at-rest with server-side encryption, but currently, there is no standard way for users to selectively encrypt files before uploading them to M-Files [46]. Thus, client-side encryption would be a new feature in M-Files. For a customer, client-side encryption provides a way to store highly sensitive documents on M-Files, while keeping them confidential from M-Files itself. The implementation of the feature is, therefore, interesting from a business point of view. It can lead to new customer use cases and potentially open new customer segments. Equally well, the implementation poses interesting challenges for a thesis, requiring the correct application of security best practices in the context of a new feature's design process, where security is a critical priority.

1.2 Contribution

This thesis is constructive research that begins with a hypothesis that client-side file encryption can be integrated with the web-based client software of M-Files while adhering to a clear notion of security and fulfilling implementation requirements. Namely, the confidentiality of encrypted files must be protected from M-Files itself, while allowing shared access to the encrypted files among a group of users. Furthermore, customers must have administrative control over encryption keys and their users' access to encrypted files.

The M-Files Confidential Document System (MFCDS) is proposed. MFCDS is a client-side file encryption system, that utilizes a web API for external key management with a trusted key management system (KMS). The API provides the required cryptographic operations to enable authenticated users to create and access encrypted files using their personal keys. MFCDS utilizes public key infrastructure (PKI) to allow users to share encrypted files with a select group of other users. MFCDS is a hybrid encryption system; public key encryption and symmetric encryption are combined with a methodology similar to established encryption systems [33][52]. A proof of concept implementation for MFCDS is implemented. The viability of the implementation is evaluated by measuring its performance.

1.3 Thesis Outline

This thesis begins with a review of related works in chapter 2. In chapter 3, an overview of client-side encryption is given. Relevant concepts are defined to accurately describe the problem domain of the implementation. Chapter 4 describes how the problem domain relates to M-Files as a product. In chapter 5, an implementation plan is described and proposed. Chapter 6 details the design and implementation of the key management web API. Chapter 7 details the implementation of changes to the client-side application. Chapter 8 is an evaluation of the performance of the implementation. In the final chapter 9, the conclusions are presented.

2. RELATED WORK

Client-side encryption, the underlying cryptography, and its applications have been the focus of extensive research. With the increasing popularity of the cloud computing model, the topic has garnered more interest as there are new challenges and avenues for innovation. Still, the basic functionality of client-side encryption relies on fundamental, time-tested concepts, such as symmetric and asymmetric encryption. For using encryption in cloud applications with multiple clients, the importance of robust key management is magnified. This chapter is a review of research on client-side encryption, related cryptography, and works on key management.

2.1 Advanced Encryption Standard and Client-side Encryption

The Advanced Encryption Standard (AES) is a symmetric encryption algorithm. It is recommended by the National Institute of Standards and Technology (NIST) and widely used in the industry today [14, p. 25] AES supports keys of 128, 192, and 256 bits in size. [30] AES is considered to be very secure. There is a theoretical increase in security between the different key sizes, but the greater efficiency with a smaller 128-bit key has made it the most common option. [7, p. 59, 65] A notion of security to describe AES is semantic security, as defined by Micali et al. [66]. A cryptosystem is said to be semantically secure if, for any given ciphertext, an adversary can conduct no computation that can produce information about the plaintext that is more accurate than random guessing [66, p. 417]. Simply put, the ciphertext contains no information about the underlying plaintext. Indeed, the only known attacks against a secure AES implementation are side-channel attacks, where additional data leaked into the environment such as noise produced by the encrypting device, are exploited, instead of the algorithm itself [83, p. 362].

Being a widely adopted and highly trusted algorithm, AES has been used in existing constructive research for client-side encryption. Feldman et al. proposed a client-side encryption framework for collaboration on documents encrypted on the client-side and stored on the backend of an untrusted cloud service provider (CSP) [32]. Similar systems have been proposed by Kanezaei & Hanapi, Kadam & Khairnar, Hosam & Hammad Ahmad, and Orobosade et al. [51][55][68][42]. These proposals use AES to encrypt data on the client-side before it is uploaded to the cloud. Recipients share the symmetric key to the data using public key cryptography. With this shared key, the data can again be decrypted when it is downloaded to the user's device. The choice of the AES algorithm in these works is attributed to its security and efficiency even with larger blocks of data [68, p. 29][55, p. 60][51, p. 52][42, p. 242].

2.2 Key Sharing for Client-side Encryption

Since many cloud services allow collaboration and include sharing features, the implementation of sharing data between users combined with client-side encryption is interesting. Because the data is encrypted, a decryption key must be shared. The proposals examined above use a hybrid encryption model. Data itself is encrypted using AES due to its higher efficiency compared to asymmetric schemes. To share the symmetric key over an untrusted CSP, these proposals use asymmetric cryptography. [68][51][55][32][42] Public keys of asymmetric encryption key pairs can be shared with other users and even the untrusted CSP. The symmetric AES encryption key can be protected from the CSP, by encrypting it with the public key of some user. This allows the owner of the corresponding asymmetric key pair to then download the encrypted data and the encrypted AES key. The owner can then use their private key to recover the AES key and use it to recover the plaintext data.

For example, the framework proposed by Feldman et al. allows multiple users to share and collaborate on documents encrypted by one user with AES. To allow a group of other users to decrypt the document, the AES key is encrypted with the public key of each user. These encrypted copies of the encryption key, one for each user, are appended to the encrypted document. The document with the attached ciphers of keys can then be stored with the untrusted CSP. Now when the user downloads the encrypted document to their client, they receive a copy of the symmetric encryption key, which they're able to decrypt with their private key, thus also granting them access to the document. To revoke a user's access to the document, the document must be encrypted again with a new key, which is again stored encrypted with the updated list of users' public keys. [32]

The Rivest–Shamir–Adleman algorithm (RSA) [75] is often chosen as the asymmetric algorithm for sharing keys [51][55]. While RSA is inefficient, it is efficient enough for encrypting the relatively small AES key [51, p. 53]. However, asymmetric encryption using Elliptic-curve Cryptography (ECC) may be considered a better option, since security is improved even with smaller key sizes and efficiency is better. Newer proposals have chosen to use ECC for this reason. [68, p. 27][42, p. 242]

2.3 Searchable and Homomorphic Encryption

One major theme that appears to be of particular interest to researchers currently is computations on encrypted data [29, p. 27]. With conventional methods, encrypted data at rest cannot be searched or otherwise operated on in a meaningful way on the server-side without decrypting it; an untrusted cloud service can run no operations on data encrypted on the client-side. However, one major benefit of cloud computing is the outsourcing of computational resources to a third party [65][78, p. 7]. Therefore, it would be ideal to be able to run computations on encrypted data, without being revealed, ideally, anything about the underlying plaintext. Untrusted cloud services could then be used for processing confidential data.

Searchable encryption allows queries to find matches within cipher text [17][36][24][28]. In addition to searchable encryption, homomorphic encryption schemes allow even arbitrary computations [86, p. 124][35, p. 99]. Since semantic security, such as provided by AES, ensures there is no information about the plaintext in the ciphers, how could an equally secure algorithm be searchable? Or perhaps more importantly, if information is allowed to be present in the ciphertext, to what extent is its security compromised? Much recent research about client-side encryption revolves around this exact problem.

Bellare et al. proposed deterministic encryption to allow conducting database queries for data encrypted on the client-side [17]. The proposal involves associating a block of encrypted data with deterministically encrypted tags. A server can then build an index to allow the client to query for the tag. Whenever the client wants to query for that block of data, it can choose the desired plain text tag, encrypt it with a deterministic scheme and send the cipher to the server. Due to determinism, the tag will result in the same cipher as when the tag was created as well as with every following query. [17, p. 13] This proposal allows search queries for encrypted data with similar efficiency to conventional methods [18, p. 3]. However, the scheme is lacking in confidentiality, because information about the equality of the tags is leaked [48, pp. 72–73]. More sophisticated schemes have been proposed by Goh, Chang & Mitzenmacher, Curtmola et al., and many others [36][24][28][69, p. 3]. These schemes utilize masking to prevent the equality of queries from leaking and thus achieve better confidentiality than deterministic encryption. [69, pp. 10–11] While such schemes do not leak equality, they can still leak additional information, making them less than semantically secure. A notion of security can be defined that permits some leakage of attributes, namely about the query patterns and tag frequency per block of data. [69, p. 14]

Fully homomorphic encryption schemes are semantically secure while allowing various computations on ciphertext [86, p. 124][35, p. 99]. With fully homomorphic encryption, computations can be conducted on the ciphertext that have a desired impact on the underlying plaintext, which can be reduced to addition and multiplication operations, allowing the system to perform any arbitrary computation. The trade-off, however, is substantially worse efficiency, in exchange for functionality and security. [29, pp. 29–30] [7, p. 17] Popa et al. proposed CryptDB, a framework that combines homomorphic encryption with searchable encryption schemes, producing a system that allows running SQL-queries on an encrypted database with provable efficiency [70].

Clearly, searchable and homomorphic encryption can be useful. However, the application of these schemes for client-side encryption appears challenging. Their practical, efficient, and secure implementation, including required server-side changes, would cause a substantial increase in implementation complexity. Since this thesis is not strictly focused on such functionality, the additional work required to utilize searchable or homomorphic encryption is considered outside the current scope.

2.4 Key Management in Cloud Encryption Systems

The strength of an encryption system relies on the proper management of associated keys. Key management entails managing the storage, access, and life cycles of keys and any related information. A system that implements this functionality is a key management system (KMS). [16, p. 1, pp. 12–13][82, p. 137] A KMS thus enables the secure use of keys by their intended users. Key management is a multi-faceted problem that has lead to the development of related standards and recommended practices. NIST has published a three-part guideline for general key management and a framework for designing cryptographic key management systems [16][13][14][11].

The OASIS organization has defined the Key Management Interoperability Protocol (KMIP), a generic protocol for key management. This protocol defines interfaces for standardized communication between key management servers and clients, enabling the exchange of cryptographic objects such as keys and certificates across a network. [54] Using a protocol like KMIP enables application developers to interface with a KMS in a standardized manner. Some cloud computing platforms offer key management as service with their own key management protocol implementations. In a book on applied cryptography, Haunts (2019) demonstrates the use of Azure Key Vault, a cloud-based KMS provided by the cloud computing platform Azure [41]. Another cloud-based KMS provided by Amazon Web Services is used by Campagna & Gueron in their proposal for a cloud-scale KMS [21]. Key management as a cloud service brings the benefits of cloud computing to key management applications, including faster development times, lowered costs, and automatic provisioning of resources [78, p. 8]. Moreover, expensive hardware protection can be used for a fraction of the cost of a self-hosted equivalent [41, pp. 145–146][21, p. 5].

Chandramouli et al. in their NIST publication further expand on key management issues specific to cloud computing [23]. They recommend an architectural solution that resolves key management issues particular to the Software as a Service (SaaS) model of cloud computing with an enterprise cloud consumer. This solution is a reverse proxy that sits between the CSP and any user traffic, providing encryption and decryption for outgoing and incoming data, respectively [23, p. 24]. Somewhat similarly, Fahl et al. proposed a system to mitigate key management related issues by isolating encryption operations from user environments entirely, instead using a dedicated encryption service with internal key management based on policies for authenticated users [31]. These solutions fit an enterprise use-case where the customer may operate an on-premises private cloud. Since users are not required to access keys, key management becomes simpler. Key management is only required for use by the reverse proxy or the dedicated encryption service deployment, which are always contained in the tightly controlled private cloud.

This thesis is focused on implementing client-side encryption from the point of view of an enterprise customer. The option of a customer-managed KMS is considered, to enable the customer to own the keys of their users. Customers may host a KMS on their own premises or trust some other cloud service for key management. A customer-owned KMS can be interfaced with via a key management protocol.

3. OVERVIEW OF CLIENT-SIDE ENCRYPTION

Several different online services implement client-side encryption. Clearly, there are a variety of use-cases and issues that can be addressed with this encryption system. This chapter examines client-side encryption in detail to accurately describe the term. The problem domain of client-side encryption implementations in modern web applications is examined. Issues that need to be solved to implement client-side encryption for a SaaS web application are described. Finally, alternative encryption systems, which may be implemented to solve similar problems, are discussed.

3.1 Client-side Encryption as a Security Solution

Encrypting user data at rest is an important recommended default practice today [78, p. 119][82, p. 228]. CSPs often protect the confidentiality of user data with server-side encryption. With server-side encryption, the data is encrypted by the CSP after it has been uploaded by a cloud consumer, as depicted in figure 3.1 [82, p. 228]. The encryption procedure, including key management, is handled by the backend infrastructure of the CSP, and the data is decrypted as required [78, p. 124]. Server-side encryption thus requires the cloud consumer to trust the CSP to properly secure access to their data [82, pp. 195–196, p. 228].

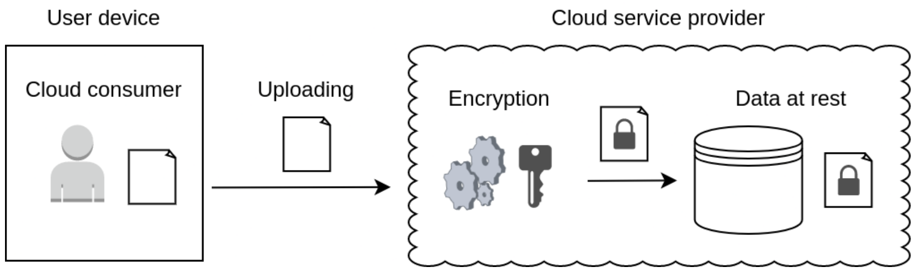


Figure 3.1. *Server-side encryption*

Unlike server-side encryption, with client-side encryption, data is encrypted before it is uploaded to a CSP. A basic overview is depicted in figure 3.2. The data is encrypted on the user's device, by client-side software, with a user-owned key. [71, p. 129][84, p. 402][43, p. 211] Therefore, the confidentiality of the data is protected from the CSP itself; the CSP can never decrypt the data since it does not know the decryption key.

For this reason, client-side encryption does not require the cloud consumer to trust any confidentiality claims made by the CSP [82, pp. 195–196, p. 228].

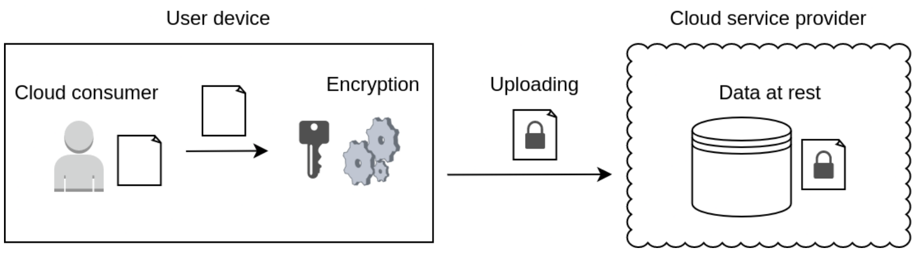


Figure 3.2. *Client-side encryption*

While CSPs have a clear incentive to protect the data of their customers, there are still many concerns with trusting a third party for data confidentiality. Firstly, aside from regular auditing, the cloud consumer may not always truly know what kind of security practices the CSP has in place. The cloud consumer must trust the claims of the CSP. There is also the threat of a malicious insider; someone within the organization of the CSP exploiting their position to gain access to the cloud consumer's data. [82, p. 55][79, pp. 3–4][25, p. 2] If the malicious insider has sufficient credentials to access encryption keys, allowing them to recover plain text data, server-side encryption is insufficient to protect confidentiality [82, p. 228]. Additionally, legal means by which user data may be leaked to unauthorized parties can be concerning to a cloud consumer. The CSP may be required to reveal user data to authorities in countries with varying data privacy laws, leading to uncertainty about the true level of privacy enjoyed by the cloud consumer [78, pp. 52–53][82, p. 53][22, pp. 44–46].

These threats can be mitigated by using client-side encryption. Assuming strong encryption is used, with client-side encryption, the CSP cannot decrypt user data. Thus, confidentiality is protected regardless of what the CSP does with the data. However, it is important to note that client-side encryption does not mitigate the threat of loss of availability, which is likewise a serious concern. This is illustrated by the recent history of ransomware attacks. [82, p. 146][6] Neither does client-side encryption address threats on the side of the cloud consumer. For example, the client-side software, where sensitive data is encrypted, can be vulnerable. Similarly, any key management system that may be used can be a point of failure.

In general, encryption alone is entirely insufficient for protection against all threats. It should be considered a complementary but vital measure as part of a whole system of protection. [78, pp. 119–120] Client-side encryption can be used to protect the confidentiality of data at rest when stored on a third party CSP.

3.2 The Problem Domain of Client-side Encryption Implementations in Web Applications

Client-side encryption protects the confidentiality of user data. Confidentiality is ensured by encrypting data on the client-side application, with a key that is not known and never revealed to the CSP. The implementation should not have functionality that relies on trust in the CSP. The client-side application, however, must be trusted to allow it to encrypt plaintext data and access any related cryptographic keys. For example, it can be assumed that the client-side application does not include any malicious code [84, p. 406]. In the case of a web application, this code is run in the user's browser.

This section describes the problem domain of a client-side encryption implementation in a web application. The cloud service model is SaaS, where a CSP serves a web application to users and allows the users to upload data and access it again later [65, p. 2]. For the sake of simplicity, in the rest of this section, the word “server” is used to refer to arbitrary backend infrastructure of the CSP. Data is encrypted in the client-side code of the web application prior to uploading to the server. When data is later fetched by recipients, users with whom the data is intended to be shared, they must be able to decrypt it again. Thus, a decryption key must be shared, all the while keeping it confidential from the server.

3.2.1 Algorithm Implementations in the Client-side Application

To encrypt on the client side, the web application must contain secure implementations of the required cryptographic algorithms. Web browsers provide Web Crypto, a standard API providing various cryptography-related functionality. The API is implemented by the browser, and exposes algorithms so that they can be called via JavaScript [39, pp. 959–960] For example, Web Crypto provides implementations of AES and RSA. Via the provided API, cryptographic keys can be generated or imported and then used in cryptographic algorithms. The supported algorithms depend on the type of key. For example, AES keys, being symmetric keys, support encryption and decryption operations. Various cipher modes are supported for AES, importantly, authenticated encryption, to protect the integrity of encrypted data. ECC is likewise supported. Web Crypto enables using the Elliptic-curve Diffie–Hellman (ECDH) protocol for sharing a secret between participants [44]. Using the ECDH implementation provided by Web Crypto, the shared secret can be derived into a shared AES key. [63] The performance of Web Crypto has been found promising, likely making it viable for most web applications [62, pp. 195–196].

3.2.2 Managing and Using Cryptographic Keys in a Client-side Application

To enable encryption and decryption in the client-side application, cryptographic keys must be generated or imported. It should be noted that the security-critical nature of cryptographic keys warrants caution. Where keys are stored and how they are used have implications for the security of the whole encryption system. Ideally, keys would only be generated, stored, and used within hardware security modules (HSM) [15, p. 11]. If encryption or key generation is done within the client-side application, this recommendation is violated. Another option would be to use a web API to request some external service to handle encryption, where keys are stored and used within HSMs. This means there would be no keys present in the client-side application at all, avoiding the problem altogether. However, encrypting via some external service would require transmitting potentially large amounts of data in requests, which could have a significant performance impact. Using the user's own device for encryption can lead to better scaling and a simpler implementation, but the security of such a system appears lacking.

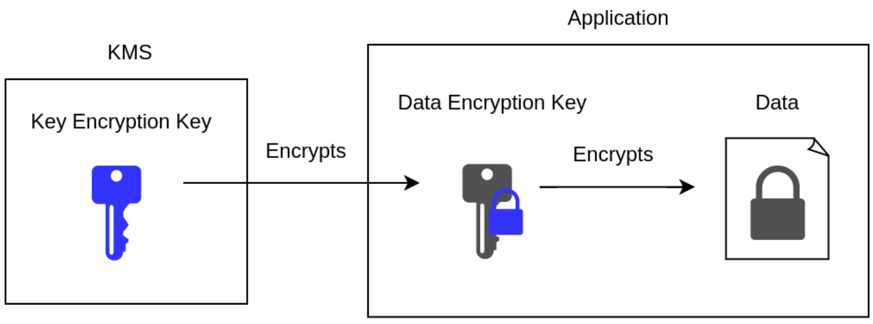


Figure 3.3. *Envelope encryption, adapted from [8]*

Envelope encryption: Envelope encryption allows the use of cryptographic keys in a client-side application while protecting them by encryption with another key stored strictly within a secure KMS. The client-side application can generate and use cryptographic keys for encryption and decryption. These keys are called data encryption keys (DEK). DEKs are protected by encrypting them with a key encryption key (KEK), which is stored in a remote KMS, ideally within a HSM. Encrypting a DEK with a KEK is known as wrapping. By wrapping the DEK, its confidentiality is protected, and it can be stored even on the untrusted server of the application. When the wrapped DEK is again used in the client-side application, it must first be unwrapped by requesting a decryption operation from the KMS. See figure 3.3 for a basic overview. [27][8] To enable envelope encryption, the KMS must implement these two operations. There must also be some way for the client-side application to request these operations, such as a web API, for example.

3.2.3 Convenient Management for Personal Keys of Users

For a user of the web application to have ownership of encrypted data intended for them to access, they must have ownership of a personal key. Technically, each user could store their key on their own device and manually enter it into the client-side application when encrypting or decrypting data. From a user experience point of view, this leaves room for improvement. Security issues arise as well since this places the responsibility of secure key storage on the user, who is likely ill-equipped for the task. Furthermore, keys should ideally be stored within a HSM, as mentioned before [15, p. 11].

Password-based key derivation function (PBKDF): To enable users to own cryptographic keys without bearing the burden of their secure handling and storage, a PBKDF can be used. A PBKDF is a function that deterministically derives a cryptographic key from a password. Typically, the user already holds one secret, their password to the SaaS application in question. If the encryption key is derived from the password, the client-side application can regenerate the key by running the PBKDF with the user's password as input and the user does not have to store an additional secret. However, when the user changes their password, all data encrypted with the key must be re-encrypted. This solution, therefore, relies on the user not forgetting their password, or else the encrypted data may never be recovered. Furthermore, user-chosen passwords are likely to lack sufficient entropy to be a safe source for generating cryptographic keys. An attacker could run attacks against the PBKDF and quickly discover the correct key if the source password is too short or easy to guess. With a complex password, high in entropy, using a PBKDF is a viable method of generating secure keys. [81, p. 5, 11] The client-side encryption systems of MEGA and SpiderOak, two cloud-based storage services under the SaaS model, both utilize PBKDFs [80][10, p. 3].

External KMS: Another option for user key management is to store the private keys of users in a trusted, external KMS and access them via a web API from the client-side application. Using an external KMS for key management allows ensuring confidentiality from the cloud service, without relying on the user's abilities to securely store their key or having to build a more intricate system like one based on PBKDF. Private keys of users can be stored in the KMS. The required operations for the keys can be exposed to the client-side application via a web API. For example, Google Workspace's implementation of client-side encryption allows a number of options for external key management. Via the provided web API, the customer can use their own KMS or use a service provided by Google's partners. [37] Using an external KMS via a web API also has the benefit of enabling the use of envelope encryption, as discussed previously. The client-side encryption system of Google Workspace utilizes envelope encryption [27]. Operations for wrapping and unwrapping the DEK are provided by the specified web API [38]. The downside of this type of client-side encryption system is that it requires another trusted service, where as a system utilizing a PBKDF can be self-contained by the SaaS application.

3.2.4 Shared Access to Encrypted Data

To allow users to share access to data encrypted on the client side, the intended recipients must be able to decrypt the ciphertext data, encrypted by another user, after fetching it from the server. Importantly, the confidentiality of the data and cryptographic keys must still be protected from the server. For example, the server cannot be used as a trusted channel for distributing keys to other users. However, the server could be used to share keys if the confidentiality of the keys is protected by encryption. Intuitively, public key encryption seems useful here. Indeed, this has been proposed by various client-side encryption frameworks (see section: 2.2), where a symmetric encryption key is protected by encrypting it with the public key of a recipient. The wrapped symmetric key can then be shared even via the untrusted server and distributed to recipients. [32] This system is also used by OpenPGP to allow multiple recipients to decrypt an encrypted email [33, p. 17]. This combination of asymmetric and symmetric cryptography is called hybrid encryption. Hybrid encryption allows combining the efficiency of symmetric algorithms such as AES with the ability to protect the confidentiality of data when it is shared on an untrusted medium. [41, pp. 113–119][51] Several cloud services with client-side encryption, too, leverage hybrid encryption for the same purpose [43, pp. 215–216][84].

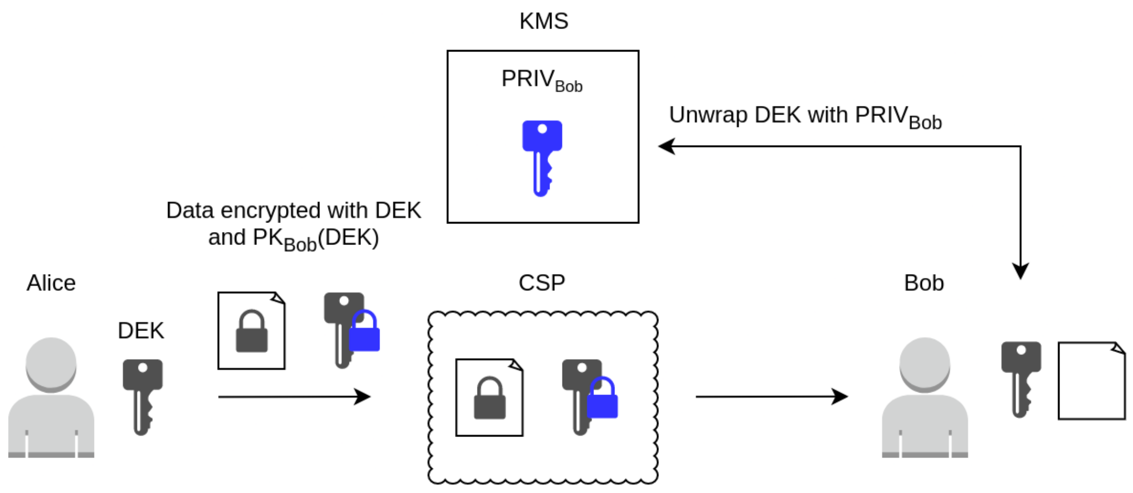

KMS
$PRIV_{Bob}$
Unwrap DEK with $PRIV_{Bob}$
Data encrypted with DEK and $PK_{Bob}(DEK)$
Alice
DEK
CSP
Bob

Figure 3.4. *Envelope encryption with asymmetric key pairs*

Envelope encryption with a remote KMS works as a key management solution when hybrid encryption is used in the client-side encryption system. Users' private keys can be stored in a remote KMS. Public keys and private can be used for wrapping and unwrapping, respectively. See figure 3.4 for an overview. In the figure, Alice encrypts data with a symmetric DEK generated on her client. She shares the encrypted data and the DEK with Bob over the untrusted CSP, by wrapping the DEK with Bob's public key. This allows Bob to retrieve the plaintext DEK by requesting an unwrap operation from the remote KMS, where his private key is stored. For users to verify the authenticity of each other's public keys, they are dependent on existing PKI, specifically, a certificate authority (CA) to verify that a public key belongs to the claimed user [20, p. 53]. Wilson and Ateniese investigated the confidentiality claims made by a number of cloud providers offering client-side encryption [84]. They found that due to the untrusted cloud provider fulfilling the role of a CA, the true level of confidentiality enjoyed by the users was doubtful. The cloud storage provider can claim its own public key to belong to one of the privileged users, thus allowing it to retrieve the shared decryption key. [84, pp. 406–407, p. 411] Therefore, the user must trust that the server is cooperative and not actively malicious. Alternatively, some other trusted service can be used as the CA.

3.3 An Alternative Approach Using a Dedicated Encryption Service

In the previous sections, the option of external key management via a web API was mentioned. This solution moves some of the cryptographic operations outside the client-side application. For example, the wrapping and unwrapping operations used in envelope encryption via a web API imply that the KMS implementing the API can encrypt and decrypt the DEK. If an external service can be practical for such limited cryptography, would it be possible to move the handling of all client-side encryption operations into an external service altogether? Especially for enterprise customers, this solution can allow leveraging the company's private cloud (see section 2.4). It would be reasonable to utilize the company's private cloud for hosting a dedicated encryption service. It may even be assumed, that the end-user environments, such as the personal devices of the company's employees, are considered less trustworthy, and certainly less secure, than an on-premises service. In this case, instead of client-side encryption, an alternative encryption system deployed in some trusted environment can be used for the same purpose of protecting the confidentiality of company data from the CSP used for data storage.

A dedicated encryption service deployed on the private cloud of a company has benefits for key management. Both the KMS and the encryption service can be contained within the company's private cloud. Instead of having to manage keys for multiple users, the service could even use a single key for all operations. Security guarantees are better because no keys must ever be exposed to uncontrollable and potentially compromised user devices and networks. A proxy service can be used, that encrypts outgoing data before it leaves the internal company network and decrypts data coming in. [23, p. 24][78, pp. 124–125] Alternatively, the encryption service could be a cloud service, that allows authenticated users to encrypt and decrypt data according to configured access control policies. User key management is handled entirely within this service. Its internal access control allows sharing access to encrypted data, without requiring the users to share keys across the untrusted CSPs application. This model of cloud service has been proposed as Confidentiality as a Service and Encryption as a Service. [73][31]

A remote encryption service was first proposed in 2001 by Berson et al. [19]. They proposed cryptography as a network service to address the issue of limited computing power of end-user devices, relevant at the time. This benefit could still be relevant today, depending on the size of the data, the computational power of the devices of end-users, and the number of requests the encryption service has to handle. In a modern client-side web application, however, cryptographic algorithms on user devices promise good performance in most use-cases [62, pp. 195–196]. Especially in the cloud computing context, as opposed to user device limitations, performance bottlenecks are often caused by heavy usage load on the backend infrastructure, requiring the provisioning of more physical resources to solve. Instead of efficiency, the better security guarantees of a dedicated encryption service should be considered its greatest benefit. Still, a solution such as envelope encryption in a client-side encryption system can be considered a good compromise (see section 3.2.2). The most sensitive keys are protected in an external KMS, and similarly, never exposed to end-users.

4. CLIENT-SIDE ENCRYPTION FOR M-FILES

M-Files is a document management platform primarily used by companies to digitalize their internal document management. M-Files can be deployed on customer premises as well as used as a cloud service. It has client software for Microsoft Windows and mobile platforms, as well as a web application, M-Files Web (MFW). All client software, including MFW, are served by the same backend infrastructure, henceforth referred to as M-Files Server (MFS). This thesis implements a proof of concept client-side file encryption system as a new feature for MFW. The basic functional requirements of the implementation are that users of M-Files must be able to encrypt files on the client side and share them with a group of other users. All the while, the confidentiality of the files must be protected from MFS. In this chapter, some basic concepts of M-Files are explained, to define more specific implementation requirements. This includes concepts related to backend logic, as implemented in MFS, as well as user interface considerations, as implemented in MFW.

4.1 Documents in M-Files

Files are stored in M-Files as documents. A document can have one file or many files (multi-file document). Each document has a title, metadata, version history, and permissions attached to them. A document's metadata can contain multiple properties of different types such as numeric values, text properties, date and time values, etc. Properties can even refer to other documents or objects within M-Files. Whenever a document or its metadata is modified by a user, a new version of the document is created. The old version is preserved in version history, and it can be easily recovered later. The permissions associated with the document dictate which users can view, modify and delete documents. Documents are stored in M-Files vaults. A customer may have multiple vaults with different configurations. The vault configuration includes, for example, settings for the different clients, authentication method configuration, and configuration of user accounts. [58][57] Documents in M-Files, therefore, are a lot more than just file storage. M-Files is a complex document management system, a comprehensive description of which is well outside the scope of this chapter.

The file part of the document can be thought of as its content, in addition to all the metadata that is associated with it. M-Files documents can store arbitrary types of files, for example, plain text files, pdf-documents, or photos and videos. It is important to note that client-side file encryption only encrypts the contained file, not any other data associated with the document. Encrypting the title of the document or some other metadata on the client-side are outside the scope of this implementation. A functionality such as client-side metadata encryption would be interesting because metadata is a widely utilized concept in M-Files, and using it to store confidential information might open interesting use-cases. However, metadata must remain functional. MFS can, for example, build indexes from metadata fields and have complex rules based on metadata values. Therefore, this type of implementation would warrant the use of searchable or homomorphic encryption (see related works in section 2.3), and further examination of how such functionality could be supported by MFS.

Since documents can store arbitrary types of files, storing ciphertext files is simpler than storing ciphertext metadata. While M-Files does have support for server-side functionality based on file content, it is not assumed that every file must be readable by the server. For example, MFS supports indexation based on file content, but the feature is only enabled for certain file types. [59] A file of any format can be stored and thus there are no assumptions that file contents need to be readable by MFS in some way. Therefore, files encrypted on the client side can be stored as ciphertext, without breaking functionality. Documents that contain files, which were encrypted on the client side, are henceforth referred to as confidential documents. Confidential documents are defined as M-Files documents, with encrypted file content, that is kept confidential from MFS.

4.2 Access Control

MFS implements a comprehensive access control system to allow for restricting users' access to documents. For example, document-specific permissions can be used to define the access rights of a user or a group of users [58]. Under the normal use-case, this internal access control is a sufficient guarantee to protect the confidentiality of documents from unauthorized users. However, with confidential documents, it is assumed that confidentiality requires further protection even from MFS itself. Therefore, access control, to the extent of protecting the confidentiality of files, must be implemented using cryptography. The client-side encryption system cannot rely on trust in MFS to protect the confidentiality and integrity of files in confidential documents.

With cryptographic access control, a user is prevented from viewing the file of a confidential document, by simply not sharing the decryption key with that user. Ultimately, however, MFS serves the document from cloud storage to the end-user. Therefore, the internal access control of MFS must be relied on for controlling which users can modify or delete the document. As such, MFS must be trusted and enabled to do this task correctly. There must be some way to allow the internal access control of MFS to cooperate with the access control of the client-side encryption system, to allow MFS to only give access to confidential documents to the correct users. Still, the confidentiality and integrity of the contained file is protected with cryptography, not with internal access control of MFS; The content of the confidential document cannot be viewed by unauthorized users, and if the content is tampered with, it can be noticed.

A lack of cooperation between internal access control and cryptographic access control can cause loss of access to data. For example, while only the users who can decrypt the file of a confidential document can view its contents, another user with permission from M-Files could still delete the document. Therefore, MFS must be able to read which users are able to decrypt the file, and subsequently prevent other users at least from deleting and modifying the confidential document, and perhaps from even viewing it. Similarly, MFS must ensure that all the users who can decrypt the file can also view it.

Another consequence of misbehaving internal access control can be observed with the version history system of M-Files. For each document, a version history is tracked. An older version of a document can be recovered, including the contents of any contained files. This has consequences for the implementation of access revocation with confidential documents. For example, a salary information document's file might be encrypted on the client-side with a key known to Alice, Bob, and Charlie. Charlie's role in the company changes and he should no longer be able to read the salary information contained in that file. Bob and Alice will then re-encrypt the document with a new symmetric key only known to them. As a result, the document's contents change, and a new version is created in M-Files. Charlie is not able to decrypt the file of this version with his old key. However, Charlie can find the old version from document history and still decrypt it. This too can be mitigated, with cooperation from internal access control. Whenever the cryptographically enforced permissions of a confidential document are updated, it should be possible to update the internal access control of MFS to reflect these changes.

4.3 Authentication and Client Authorization to External Services

M-Files allows administrators to configure internally managed user accounts for each document vault, allowing users to authenticate directly with MFS. However, since the confidentiality and integrity of files in confidential documents is to be protected from MFS itself, internal identity management, and the associated internal access control, cannot be relied on by the client-side encryption system. M-Files also allows configuring Azure Active Directory (AAD) as an external identity management service. [60] AAD is thus an intuitive first choice for a trusted external identity provider for the implementation. For example, the implementation can use AAD to authenticate users and authorize MFW to access a web API, that enables envelope encryption with an external KMS (see section 3.2.2).

Authentication with AAD is enabled by the OAuth 2.0 protocol extended with the OpenID connect protocol [67]. Depending on the type of application, different grant flows can be used to acquire access tokens for the user's resources. If the application has an application server, such as MFS, the plain authorization code flow can be used. This flow is designed to be used with a shared secret between the application server and the OAuth provider's authorization server. This shared secret is used to authenticate the application server, preventing an illegitimate server of an attacker from making requests on behalf of the user. [40] However, in this implementation, even the genuine application server is not trusted to access the user's resources in a way that could compromise the confidentiality provided by the client-side encryption system. For example, MFS is not trusted to use an external web API for key management on behalf of the user. As such, the plain authorization code flow is unsuitable for the implementation.

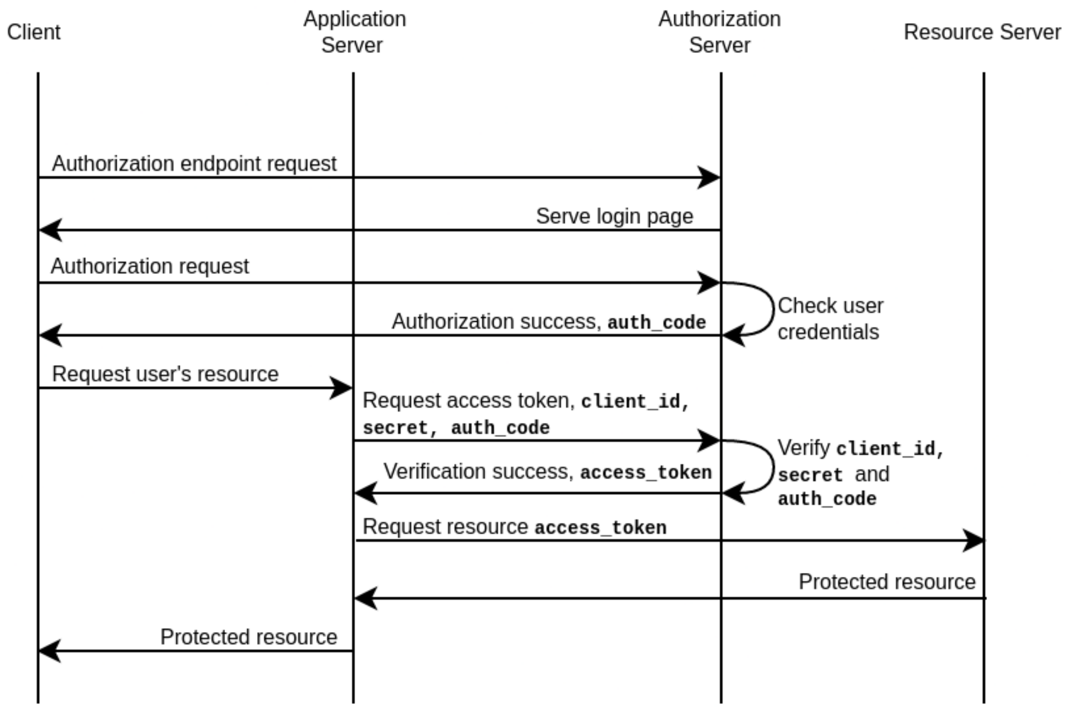

Client
Application Server
Authorization Server
Resource Server
Authorization endpoint request
Serve login page
Authorization request
Check user credentials
Authorization success, auth_code
Request user's resource
Request access token, client_id, secret, auth_code
Verify client_id, secret and auth_code
Verification success, access_token
Request resource access_token
Protected resource
Protected resource

Figure 4.1. *OAuth 2.0 plain authorization code flow*

Some other grant flow that enables the client-side application to access the user's resources must be used. The user must be authenticated with AAD and then MFW must be authorized to use any required services on behalf of the user. All the while, the resources must be protected from MFS; MFS should not be authorized to access them. The plain authorization code flow cannot be used without the shared secret, for the reasons described above. The shared secret likewise cannot be stored in a public client-side application such as MFW, as it could easily be leaked or stolen. To solve this problem, the authorization code flow can be extended using RFC 7636 protocol also known as Proof Key for Code Exchange (PKCE), which is designed specifically to allow using OAuth 2.0 with public clients [76]. PKCE introduces a simple mechanism for creating temporary

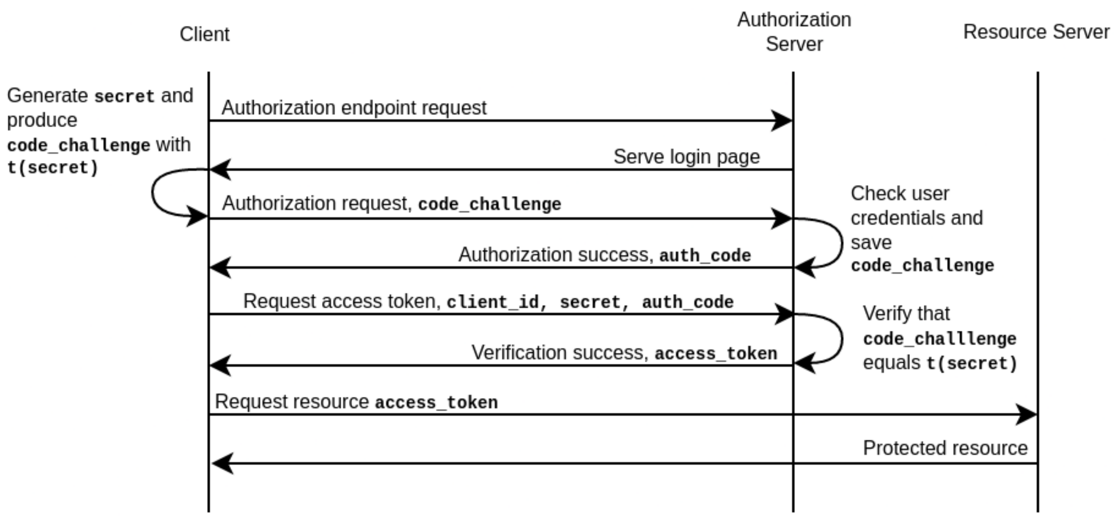

Client
Authorization Server
Resource Server
Generate secret and produce code_challenge with t(secret)
Authorization endpoint request
Serve login page
Authorization request, code_challenge
Check user credentials and save code_challenge
Authorization success, auth_code
Request access token, client_id, secret, auth_code
Verify that code_challlenge equals t(secret)
Verification success, access_token
Request resource access_token
Protected resource

Figure 4.2. *OAuth 2.0 authorization code flow extended with PKCE*

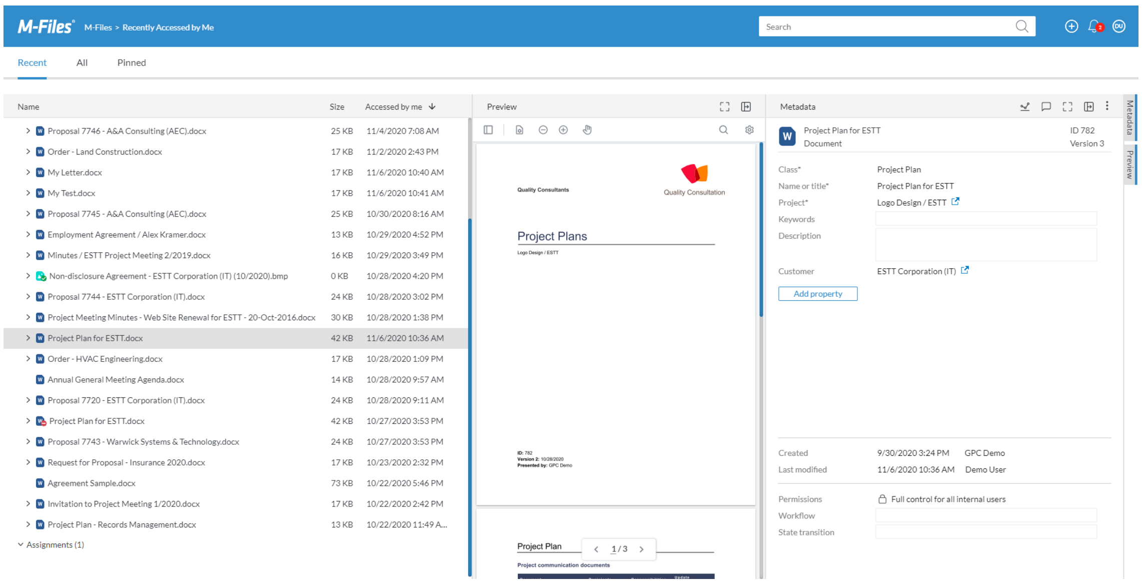
secrets to provide similar security to the plain authorization code flow. With PKCE, the client generates its own secret and transforms it into a code challenge using some known function, typically by hashing. This code challenge is sent to the authorization server in exchange for an authorization code. The next time this authorization code is sent to the authorization server in exchange for an access token, the server expects to also receive the secret that was used to create the code challenge. The server thus challenges the request, by transforming the received secret and comparing its equality to the initial code challenge it received. If the challenge fails, the request is forbidden. This mechanism prevents an attacker from abusing a stolen authorization code. [76, pp. 8–10] See figures 4.1 and 4.2 for an overview of the plain authorization code flow and authorization code flow with PKCE respectively.

Using OAuth 2.0 with PKCE allows authorizing MFW to use services on the behalf of the user, without involving MFS in the authorization flow. AAD supports the PKCE grant flow and can thus be used as the trusted identity provider [67].

4.4 User Interface Considerations

MFW is a single-page web application (SPA) that is used to browse content within M-Files. In this thesis, user interface (UI) changes are implemented to the extent that the basic functionality of the proof-of-concept implementation can be demonstrated.

Users can create new documents of different types via MFW. Similarly, users should be able to choose confidential document as the type of document to create. Upon creation, options specific to client-side encryption should be displayed to the user, such as options related to sharing the document with other users. To initiate creating a new document, the user can drag and drop a file into the browser window or click the corresponding



Name	Size	Accessed by me
Proposal 7746 - A&A Consulting (AEC).docx	25 KB	11/4/2020 7:08 AM
Order - Land Construction.docx	17 KB	11/2/2020 2:43 PM
My Letter.docx	17 KB	11/6/2020 10:40 AM
My Test.docx	17 KB	11/6/2020 10:41 AM
Proposal 7745 - A&A Consulting (AEC).docx	25 KB	10/30/2020 8:16 AM
Employment Agreement / Alex Kramer.docx	13 KB	10/29/2020 4:52 PM
Minutes / ESTT Project Meeting 2/2019.docx	16 KB	10/29/2020 3:49 PM
Non-disclosure Agreement - ESTT Corporation (IT) (10/2020).bmp	0 KB	10/28/2020 4:20 PM
Proposal 7744 - ESTT Corporation (IT).docx	24 KB	10/28/2020 3:02 PM
Project Meeting Minutes - Web Site Renewal for ESTT - 20-Oct-2016.docx	30 KB	10/28/2020 1:38 PM
Project Plan for ESTT.docx	42 KB	11/6/2020 10:36 AM
Order - HVAC Engineering.docx	17 KB	10/28/2020 1:09 PM
Annual General Meeting Agenda.docx	14 KB	10/28/2020 9:57 AM
Proposal 7720 - ESTT Corporation (IT).docx	24 KB	10/28/2020 9:11 AM
Project Plan for ESTT.docx	42 KB	10/27/2020 3:53 PM
Proposal 7743 - Warwick Systems & Technology.docx	24 KB	10/27/2020 3:53 PM
Request for Proposal - Insurance 2020.docx	17 KB	10/23/2020 2:32 PM
Agreement Sample.docx	73 KB	10/22/2020 5:46 PM
Invitation to Project Meeting 1/2020.docx	17 KB	10/22/2020 2:42 PM
Project Plan - Records Management.docx	13 KB	10/22/2020 11:49 A...

Assignments (1)

Project Plan for ESTT
Document
ID 782
Version 3

Class*	Project Plan
Name or title*	Project Plan for ESTT
Project*	Logo Design / ESTT
Keywords	
Description	
Customer	ESTT Corporation (IT)
Created	9/30/2020 3:24 PM GPC Demo
Last modified	11/6/2020 10:36 AM Demo User
Permissions	Full control for all internal users
Workflow	
State transition	

Figure 4.3. *M-Files Web user interface [61]*

icon from the top bar of the UI. In the latter case, the initial file of the document will be empty. In the former, the dropped file is immediately uploaded to MFS, after which the user is prompted to define metadata properties and a title for the document. The file of a new confidential document needs to be encrypted before it is uploaded to MFS, and thus the existing flow requires changes. Furthermore, additional options specific to creating confidential documents must be displayed to the user before encryption and subsequent uploading of files.

Files contained in documents can be viewed in MFW. A preview area is used for this, the middle panel shown in figure 4.3. To demonstrate the functionality of the proof of concept implementation, this preview area is used to view the file of a confidential document after it has been successfully decrypted.

5. PROPOSED IMPLEMENTATION

This chapter proposes the implementation plan for MFCDS, enabling client-side file encryption for M-Files. MFCDS enables users to create confidential documents, encrypted on the client side. Hybrid encryption is used; Files contained in confidential documents are encrypted with symmetric encryption, while public key cryptography is used to share symmetric keys between users. The key pairs of users are stored with an external KMS. Envelope encryption is used for wrapping and unwrapping DEKs with user key pairs. Files are encrypted and decrypted with DEKs on the client side, while the most sensitive keys of users are kept strictly inside the external KMS. The required key management operations are exposed via a web API. To access the API and use client-side encryption functionality, users are required to authenticate with a trusted identity provider.

5.1 Overview

MFCDS adds capability to the client-side application, MFW, to enable users to create confidential documents, share them with a group of users (henceforth referred to as the allowed group), and allow anybody in that group to consequently decrypt and view the document. See figure 5.1 for a high-level overview of the architecture. The file content of a confidential document is encrypted using a symmetric DEK generated on the client. The public key of each user in the allowed group is used to wrap the DEK. These wrapped DEKs are stored with the document on MFS. Each user in the allowed group can use their externally stored private key to unwrap their copy of the DEK and retrieve the plaintext file of the document.

A simple file format is defined, that is followed by the files in confidential documents. The client-side application parses the file, as dictated by this format, enabling subsequent decryption by users in the allowed group. The file format is depicted in figure 5.2. The first 4 bytes of the file contain the length of the following metadata block. This block contains copies of the DEK, encrypted with each public key in the allowed group. Any other required metadata is also to be contained in the metadata block. The metadata block is formatted as a JSON Web Token (JWT). The JWT is signed with the private key of the creator of the document, to ensure integrity and prevent, for example, a malicious MFS from tampering with the contents. While the DEK is technically a shared secret

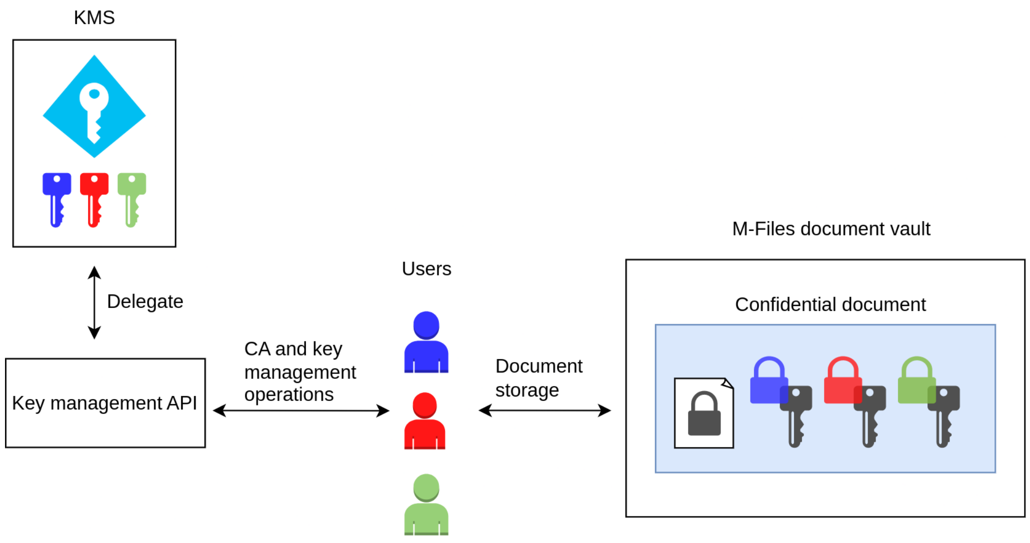


Figure 5.1. *Overview of the proposed client-side encryption system*

between users in the allowed group, it should not be used to sign the JWT. If the JWT is signed with the DEK, a malicious server could replace the contents of the file and make it appear legitimate by generating its own DEK, encrypting another file with this DEK, and re-creating the JWT accordingly, signed with the false DEK. When the JWT is signed with the creator's private key, other users can verify the integrity of the file by verifying the creator's identity with a trusted CA.

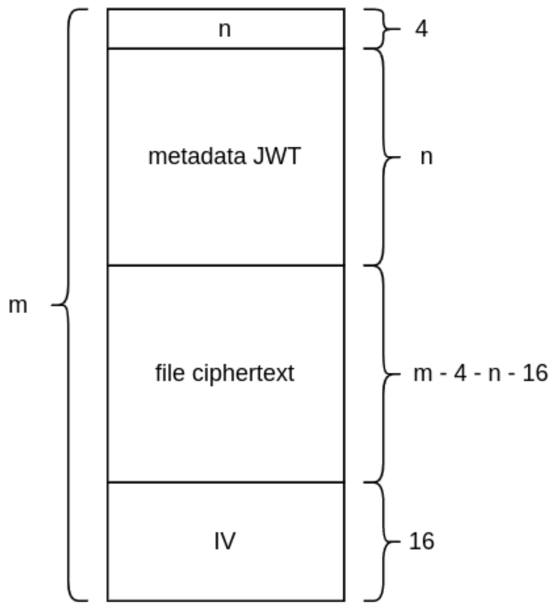


Figure 5.2. *Confidential document file format*

The metadata block is followed by the ciphertext of the file itself, encrypted using the DEK. The symmetric algorithm used with the DEK is AES with the Galois/Counter mode of operation (AES-GCM) [53]. Using AES-GCM makes the encryption operation authenticated, thus protecting integrity of the ciphertext. While difficult, it is possible to modify the ciphertext in such a way that the resulting plaintext content has meaningful changes. With AES-GCM, such tampering would be noticed upon decryption, and the file can be flagged as compromised. [7, p. 16, p. 152] Finally, the fixed length initialization vector (IV) is appended at the end of the file.

To define the allowed group for a confidential document, the user creating the document must have access to the public keys of other users. Similarly, each user in the allowed group must be able to use their own private key to decrypt their copy of the DEK. As discussed in section 3.2.3, giving users the responsibility of safely storing cryptographic artefacts such as private keys is risky. Instead, a trusted external KMS is used to store users' private keys securely. In this case, the KMS is accessed via a key management web API, which interfaces with the KMS and exposes the required operations to the client. Users are not burdened with key management. Instead, the client-side application code requests the required key management operations from the web API. As mentioned above, the user must verify the integrity of the metadata block with a CA. This operation should likewise be provided by the API. The operation essentially entails mapping public keys to user identities, so that the user can verify the identity of the owner of a public key, before using the key to verify a signature.

5.2 External KMS and the Key Management Web API

In MFCDS, user keys are stored in an external KMS that is accessed via a web API. Because the API is used to access the personal keys of users, the caller must be authenticated, and the client-side application (MFW) must be authorized to use the web API. Users authenticate with a trusted external identity provider, so as to not rely on trust in internal identity management of MFS. As discussed in section 4.3, AAD can be used for this purpose. For key management, the Azure cloud platform provides Azure Key Vault (AKV), a cloud-based KMS solution. AKV offers multiple options for secure key storage, compliant with different levels of the security standard for cryptographic modules specified by NIST [34]. The highest level of security is provided by managed HSM instances and the lowest by standard multi-tenant key storage without hardware protection. [4] AKV provides an API with the required functionality for envelope encryption [26]. Asymmetric key pairs can be stored on AKV and, via the provided API, used for wrapping, unwrapping, signing of data, and other operations. [9] While there are other similar services, like the KMS provided by Amazon Web Services [8], the easy AAD integration provided by AKV makes it a convenient choice for this implementation.

The key management API must also provide the role of CA. The role of the CA entails associating a user identity with a public key [7, p. 238][5]. The desired client-side functionality in MFW is to allow users to verify the identity of another user based on their public key. Furthermore, MFW should be able to list the public keys of other users, associated with some unique ID (a name or an email address, for example). However, AKV does not provide the functionality to allow assigning keys to individual users. Another service must be built to implement this functionality. Such a service maps the identity of a user to their key and implements the required endpoints of the key management API, by delegating calls to AKV.

The required operations for a CA as well as operations for envelope encryption can all be defined into a set of web API endpoints. Any KMS with similar functionality to AKV could be used. For example, a customer can host their own KMS or use another cloud-based service and implement the defined key management API to be compatible with it. For the proof of concept implementation, AKV is used as the KMS to demonstrate the viability of MFCDS. A web service that implements the API, delegating key management to AKV, will be implemented to enable envelope encryption as a key management solution for user keys. The API also enables the PKI required by public key cryptography of the implementation, by fulfilling the role of the CA.

5.3 Use Cases and Required Operations of the Implementation

This section examines the use cases required to fulfill the basic functionality of the implementation. Namely, users must be able to create, update, and view confidential documents. Users must also share access to the document with other users in a chosen group. These use cases rely on operations implemented by the key management API and MFW. To authenticate the user and authorize MFW to use the endpoints of the API, an identity management service is also required. The operations provided by these components enable the use cases of the implementation and outline the requirements of the key management protocol that the key management API implements.

5.3.1 Creation

Users must be able to create confidential documents. This use case requires listing the public keys of other users, to allow the creator of a confidential document to choose the allowed group of users, by whom the file of the document can be decrypted. The creator must also be able to use their own private key to sign the metadata section of the contained file. A backup of the DEK must also be created. Creating a confidential document, therefore, entails using API endpoints for a number of operations. It is assumed that the key pairs for the users already exist in the KMS, having been, for example, created by an administrator.

Diagram 5.3 describes the flow of the procedure of creating a confidential document and choosing the allowed group of users. First, the creator chooses the UI option to start creating a confidential document and consequently uploads a plaintext file to MFW, which

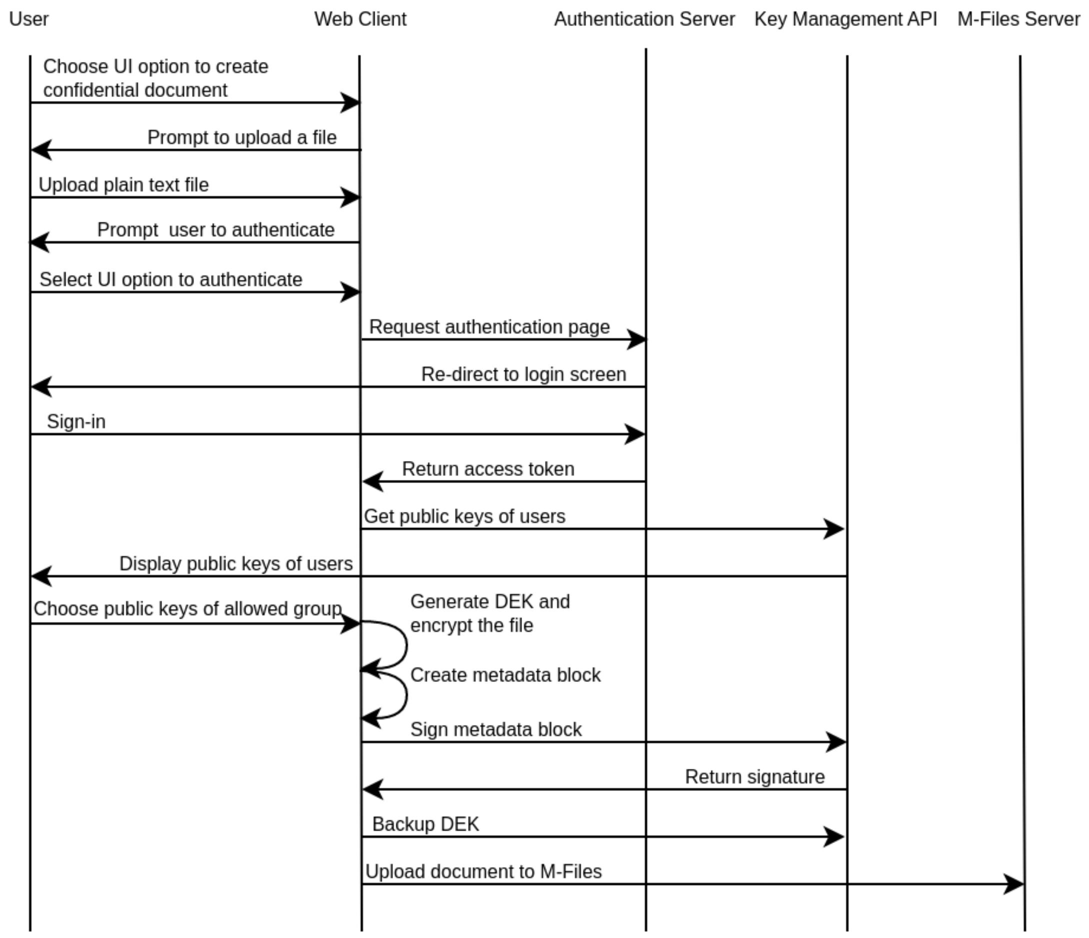

User
Web Client
Authentication Server
Key Management API
M-Files Server
Choose UI option to create confidential document
Prompt to upload a file
Upload plain text file
Prompt user to authenticate
Select UI option to authenticate
Request authentication page
Re-direct to login screen
Sign-in
Return access token
Get public keys of users
Display public keys of users
Choose public keys of allowed group
Generate DEK and encrypt the file
Create metadata block
Sign metadata block
Return signature
Backup DEK
Upload document to M-Files

Figure 5.3. *Creating a confidential document*

is stored in browser memory for the time of the procedure. Then, to choose the allowed group of users, the creator authenticates and authorizes MFW to use the API. After successful authentication, the creator's browser session is allowed access to the API and the client-side application can list the public keys of other users. The public keys are listed along with a unique user ID for each key, such as the user's email or username. After choosing the public keys and thus the allowed group, the creator initiates the encryption procedure. Client-side code generates a DEK for AES-GCM and creates encrypted copies of it by wrapping the DEK with each public key in the allowed group. These wrapped DEKs are then saved into the metadata block of the file, along with the ID of the owner of the public key. In the metadata block is also saved the ID of the creator user. This metadata block is then signed with the creator's private key, again requiring a call to the API. After signing, the file content is complete. The DEK is once more wrapped with a master public key, which is stored as a backup via an endpoint of the API. This allows an administrator to recover the file in case user keys are lost. At this point, there are no longer any plaintext copies of the DEK in memory. The confidential document is then uploaded to MFS.

5.3.2 Viewing

Users in the allowed group of a confidential document must be able to decrypt the contained file to view it. Also, the integrity of the contained file must be verified with the public key of the creator.

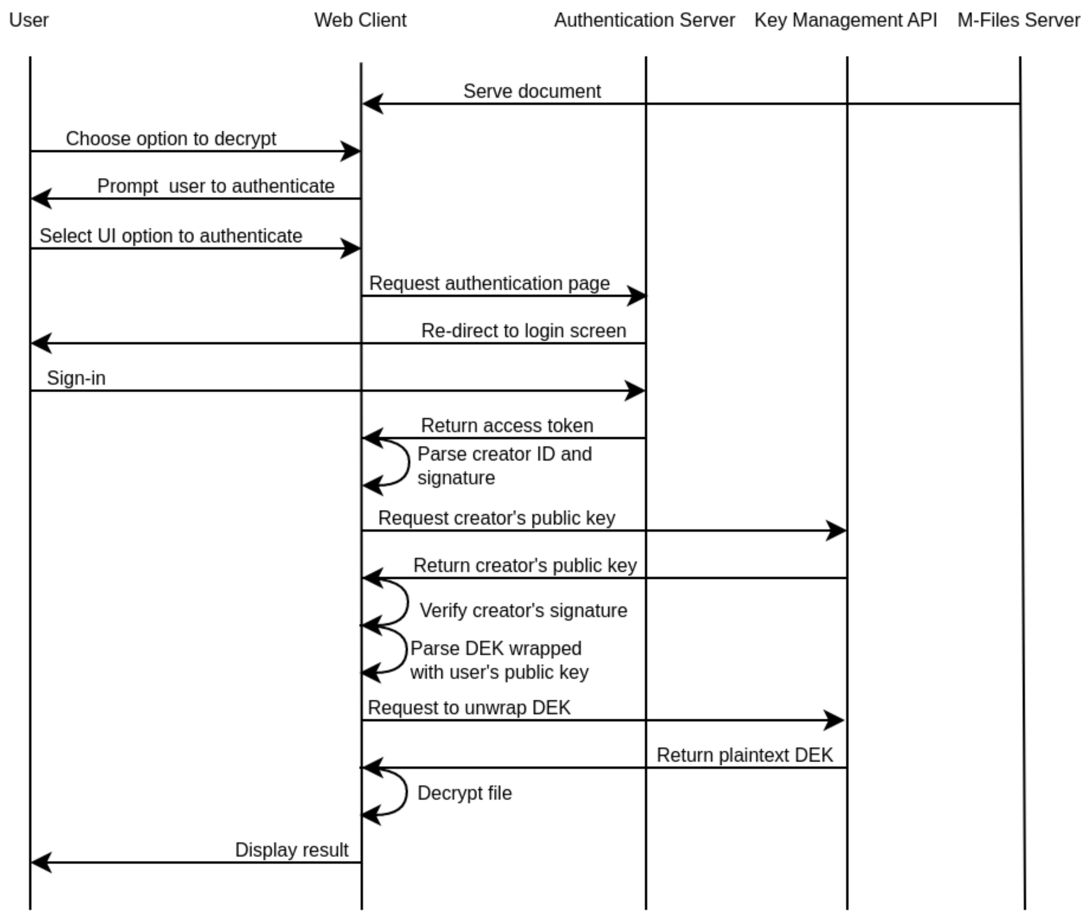

User
Web Client
Authentication Server
Key Management API
M-Files Server
Serve document
Choose option to decrypt
Prompt user to authenticate
Select UI option to authenticate
Request authentication page
Re-direct to login screen
Sign-in
Return access token
Parse creator ID and signature
Request creator's public key
Return creator's public key
Verify creator's signature
Parse DEK wrapped with user's public key
Request to unwrap DEK
Return plaintext DEK
Decrypt file
Display result

Figure 5.4. *Viewing a confidential document*

The procedure to view a confidential document is depicted in diagram 5.4. First, the user navigates to a confidential document and chooses the UI option to decrypt it. The client-side application will then begin to parse the content of the included file. To start parsing, the user is first prompted to authenticate and authorize MFW to use the API. From the metadata block, first is found the ID of the document's creator and their signature. The signature must be verified with the public key of the creator. The user requests the public key of the creator from the API and then uses it to verify the signature. After successful verification, the metadata block is considered authentic, and parsing can continue. The user then finds the DEK wrapped with their public key from the metadata block. Having been authorized already, the client-side application will now request the API to unwrap the wrapped DEK using the private key of the current user. The plaintext DEK is returned in the response. The file content is then decrypted with the DEK, and the file is displayed to the user.

5.3.3 Updating

The procedure to update a confidential document is nearly synonymous with the creation procedure. Updating entails modifying the file content or redefining the allowed group (adding or removing users), or both. If the content of the file is modified, the DEK should be re-generated. Likewise, if only the allowed group is updated, a new DEK must be generated to prevent a revoked user from using a DEK from a previous version to decrypt the newer version. In any case, because a new DEK is used for each version, both the metadata block and the ciphertext content of the file will be created again. Similarly, a new backup of the DEK should be created that is specific to this newer version. Therefore, the only differences to the procedure described in diagram 5.3 are how the procedure is presented to the user via the UI and how the document is stored in MFS. Instead of creating a new document in MFS, a new version is created.

5.3.4 Creating Backups of DEKs

In the use cases for creating and updating confidential documents, in sections 5.3.1 and 5.3.3, a backup of the DEK is created. Key backups are stored in the remote KMS, accessed via the key management API. Backups of DEKs mitigate the potential loss of data due to lost user keys. If user keys were to be destroyed prematurely and later a document was found that could only be decrypted by those keys, its plaintext content would be permanently lost.

Whenever a document is created or updated, a new DEK specific to that version of the document's file will be created, and likewise backed up. Before uploading the backup DEK via the API, it is wrapped with a public key of a master key pair, such as the key belonging to the KMS or an administrative user. This protects the DEK when it is stored, allowing retrieval of the plaintext DEK only with the private key of an administrator. To allow later retrieval of the correct backup DEK, in the metadata block of the file will be saved the universally unique identifier (UUID) of the file version. The backup will be stored in the KMS, indexed by this UUID.

5.4 Evaluation of the Proposal and Discussion on Further Improvements

MFCDS aims to address all the issues and requirements that were examined in chapter 4. The proposed client-side encryption system enables the proof of concept implementation to demonstrate the viability of this system for further development. Further development of this system should not require significant changes to the fundamentals of the system and the basic functionality that is implemented. Still, there would no doubt be much to improve to bring the implementation up to the standard that is required from a feature used in production and shipped to real customers.

5.4.1 Hybrid Encryption

A similar combination of asymmetric and symmetric cryptography is used by OpenPGP [33]. The message format of OpenPGP likewise stores a symmetric key, wrapped with the public keys of recipients, to enable recipients to decrypt the message. [33, p. 17]. Similar systems have already seen wide adoption and, as such, are expected to provide good security. Related works, too, have proposed similar methods (see section 2.2).

5.4.2 Choice of Public Key Encryption Algorithm

The proof of concept implementation of MFCDS uses RSA as the public key encryption algorithm. The other option would be to use an application of ECC. Elliptic Curve Diffie-Hellman (ECDH) can be used in a similar protocol, to share a symmetric key between users [44, p. 10]. Indeed, an ECC-based extension to OpenPGP defines a protocol utilizing ECDH [47]. In this protocol, the procedure to wrap the DEK is different than with RSA, due to the inherent properties of ECC. Namely, ECC public keys are not used to encrypt data. Instead, ECDH requires the recipient's public key and the sender's private key to compute a shared secret, from which a shared AES key can be derived. The derived AES key is used as a KEK, which is used to wrap the DEK. The recipient then needs the sender's public key to compute their KEK and unwrap the DEK. [47, p. 5-6][44, p. 11]

In MFCDS, the user's private key is accessed via the web API. Therefore, deriving shared keys with ECDH via the API for a large number of recipients would be a very time-consuming operation. Instead, a temporary ECC key pair could be computed that is only used on the client side. The private key would only be used to create a KEK for each recipient, before being destroyed. The public key of this key pair would be attached to the metadata block, and so on, similarly to the ECC OpenPGP extension. This ECC variant of the protocol was not used, because AKV does not have an endpoint to support key derivation according to ECDH. Also, RSA keys are widely used and expected to provide adequate security, provided the minimum key size of 2048 bits is used [14, p. 12][12, p. 15]. As such, RSA was chosen for the implementation. For further development, this implementation could be adapted to support ECC with small changes, provided a compatible external KMS is used. The required changes to support ECC are also documented in implementation chapters 6 and 7.

5.4.3 Protection of Confidentiality and Integrity

The main requirement of the implementation, and possibly its greatest benefit, is the guarantee of confidentiality for encrypted data. Confidentiality is protected with hybrid encryption. A sufficiently strong symmetric encryption algorithm (AES-GCM) is used to encrypt file content. The confidentiality of the symmetric DEK is protected with public key encryption. The plain text DEK is thus never revealed to MFS. Ultimately, the confidentiality of the DEK is dependent on the strength of the public key encryption algorithm. As mentioned in the previous section, RSA is considered strong provided keys of sufficient size are used.

The integrity of the files is likewise protected with cryptography, but ultimately the protection also relies on the CA functionality provided by the key management API. Files are encrypted using AES-GCM with the DEK. AES-GCM, being an authenticated encryption algorithm, means that tampered file content will not go unnoticed, provided that the DEK also retains its integrity [53]. If a malicious MFS were to replace the DEK of the file in a confidential document, it would also be possible to create completely new file content, encrypted with the false DEK. The false file would then appear legitimate. For this reason, the integrity of each wrapped DEK is also protected by the signature of the metadata block, signed with the creator's public key. While a malicious MFS can replace the metadata block, it cannot spoof its signature. The identity of the creator is verified by the CA, provided by the key management API.

5.4.4 Alternative Solutions For Convenient Management of User Keys

MFCDS solves the problem of user key management by using envelope encryption and an external KMS. Another solution that was considered is a system that utilizes PBKDF to derive user-owned cryptographic keys from user passwords. In section 3.2.3 some merits and disadvantages of this system are mentioned. Ultimately, for this implementation, it was found unsuitable, mainly because such a system would not be compatible with the identity management system of M-Files. Not only would the implementation of a PBKDF-based system require heavy modifications to the implementation of user accounts in MFS, but it would also not work in the first place when AAD is used for identity management. Users are authenticated with OpenID Connect, and thus user passwords are never present on the client side when logging in. Therefore, envelope encryption with a remote KMS was found to be the more suitable option.

In addition to client-side encryption, a dedicated encryption system was considered. This solution was discussed in length in section 3.3. Indeed, a dedicated service for encryption would provide a better guarantee of security for key storage, because no keys ever need to be even temporarily present in the client-side program. This thesis proposes that the solution here represents a good compromise. In MFCDS, user keys are likewise contained strictly in a remote KMS and only accessed via the API. Having plain text DEKs briefly in the client-side application is thought to be an acceptable compromise. Inevitably, the file content, too, is in plain text during this time, while a confidential document is being viewed or created. This thesis hypothesizes that the ability to utilize user devices for potentially expensive cryptography with file data can lead to better scaling for the application as a whole.

6. DESIGN AND IMPLEMENTATION OF THE KEY MANAGEMENT API

This chapter details the implementation and design process of the key management API (referred to as API in the rest of this chapter). The API implements a simple key management protocol, enabling envelope encryption for MFCDS with a remote KMS. The endpoints of the API must provide operations that enable the use cases described in section 5.3. Via the API, the user can use their private key for the required operations. Similarly, the API allows the user to list the public keys of other users and get the public key of a single user.

6.1 Defining the Endpoints of the Key Management API

The API is implemented as a remote procedure call (RPC) protocol. The endpoints of the API do not always represent resources, but they can also refer to functionality requested from the KMS, such as wrapping and unwrapping keys. As such, RPC was found to be the more intuitive choice over the other common option, representational state transfer.

Common to all the endpoints is that they require a bearer access token issued by the identity management service. If no valid access token is provided, the endpoints respond with the unauthorized status code, 401. Bodies of requests and responses are in JSON format. In some cases, binary data must be transmitted. For example, in the operation to unwrap a DEK, the wrapped DEK is in binary format, but must be passed in the request body. In this case, the data is encoded as base64, allowing it to be transmitted in JSON format, like any other string value. In case the request body is in an invalid format, the status code 400 is returned to indicate this.

6.1.1 Getting a Public Key

The endpoint specified in table 6.1 allows getting the public key of an individual user to verify their signature, or to get the master public key to create a backup of a DEK. The endpoint takes one optional input parameter, which is the ID of the user. This can be, for example, an email, a username, or some other unique identifier, depending on how the API is implemented. If no user ID is specified, the API returns the master public key of the

KMS, which can be used to wrap a DEK to create a backup. The public key is returned in the JWK format [50]. JWK is a standard data structure to store multiple types of keys. For example, if an ECC variant of this protocol was implemented, the endpoint could return an ECC public key as a JWK. [50, p. 25]

By mapping a user ID to a public key, this endpoint, in effect, issues a certificate to the web-client. There is no need for using a standard certificate format for the purposes of the MFCDS. Instead, this endpoint suffices as a trusted "source of truth", for getting the public key of a user.

Endpoint: get-public-key		
Request JSON		
Property Name	Type	Description
user	String	Optional parameter. A unique identifier of the owner of a public key. If not provided, returns the public key of the KMS itself.
Successful response, a JSON object		
Property Name	Type	Description
type	String (`"user"` or `"kms"`)	Indicates if the returned key belongs to a user or if is the master key, belonging to the KMS.
key	JWK	A public key in JWK format.

Table 6.1. *The endpoint for getting a specific public key.*

6.1.2 Listing Public Keys

The API can be used to list the keys of users. This allows choosing the allowed group of users when creating or updating a document, as described in sections 5.3.1 and 5.3.3. The endpoint takes no parameters and returns an array of JSON objects. See table 6.2 for the specification. The returned keys are in the JSON Web Key (JWK) format. This endpoint could be further improved to allow a query with finer granularity, for example, by introducing a parameter to filter the result. For the proof of concept implementation of MFCDS, it suffices that the keys of all users are returned.

6.1.3 Creating a Backup of a Key

Table 6.3 specifies the API endpoint to create a backup of a DEK, as described by the operation in section 5.3.4. To create a backup of a DEK, the previous endpoint specified in 6.3 is first used to fetch the public key of the KMS. After the DEK is wrapped with this key, it can be uploaded to the API in the request body. In addition to the wrapped DEK, the endpoint requires a parameter to uniquely identify the version of the confidential

Endpoint: list-keys		
Request JSON		
None		
Successful response, array of JSON objects		
Property Name	Type	Description
user	String	A unique identifier of the user.
key	JWK	A public key in JWK format.

Table 6.2. *The endpoint for listing all public keys.*

document that the DEK belongs to.

Endpoint: create-key-backup		
Request JSON		
Property Name	Type	Description
wrappedKey	String (base64)	A DEK wrapped with a master public key.
dataUid	String	The unique identifier for the data that can be decrypted with the DEK.
Successful response, a JSON object		
Property Name	Type	Description
result	Boolean	Indicates if the backup was created successfully.

Table 6.3. *The endpoint for creating a DEK backup.*

6.1.4 Unwrapping a Wrapped Key

Each user must be able to use their private key to decrypt a DEK wrapped with their public key. This operation is offered by the endpoint specified in table 6.4. When this endpoint is called, the service implementing the API must use the private key of the calling user to decrypt the document. The ID of the user is thus to be contained in the bearer access token.

If this API were to be implemented to support an equivalent protocol with ECC using ECDH, this endpoint would have to be slightly altered. The endpoint would also require the public key of the creator user as an input parameter in the request body. It would use this public key with the private key of the calling user to run a key derivation function, producing a KEK. [47, pp. 5–6] Then, the endpoint would use the derived KEK to unwrap the DEK. See section 5.4.2 for a further description of how the implementation can be adapted to support ECC.

Endpoint: unwrap-key		
Request JSON		
Property Name	Type	Description
key	String (base64)	The DEK to unwrap. Must be wrapped with the public key of the user calling the endpoint.
Successful response, a JSON object		
Property Name	Type	Description
result	String (base64)	The plain text DEK as base64.

Table 6.4. *The endpoint for unwrapping a wrapped DEK.*

6.1.5 Signing Data

As described in sections 5.3.1 and 5.3.3, when a document is created or updated, the creator's private key is used to sign the metadata block. Similarly to the previous endpoint for unwrapping, the caller of this endpoint may only use their own private key. This endpoint for this operation is specified in table 6.5. The metadata block is signed with RSASSA-PKCS1-v1_5 using SHA-256, the recommended algorithm for JWK signatures [49, p. 6].

If ECC keys were to be used, the recommended algorithm that should be used is ECDSA with the P-256 curve and SHA-256 [49, p. 6].

Endpoint: sign		
Request JSON		
Property Name	Type	Description
message	String	The message to sign.
Successful response, a JSON object		
Property Name	Type	Description
result	String (base64)	The signature.

Table 6.5. *The endpoint for signing the metadata block.*

6.2 Implementing the API as a Cloud Application

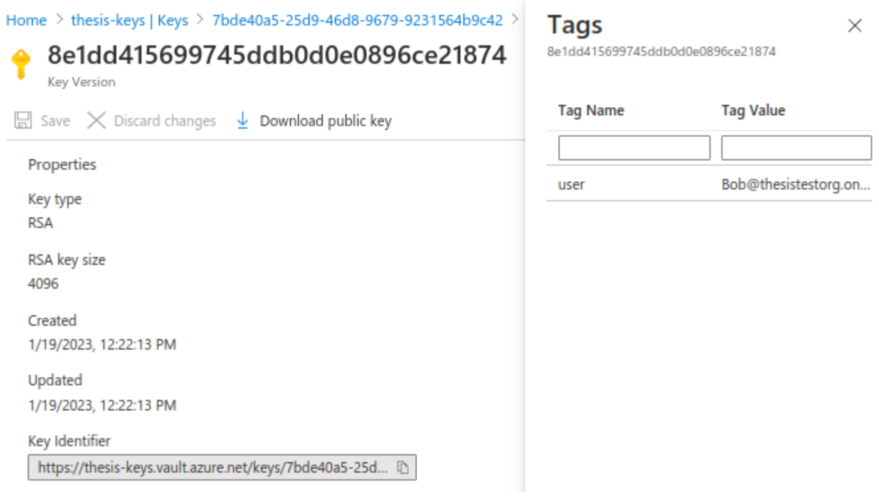
The API was implemented as a serverless function application on the Azure cloud computing platform. This option was chosen mainly for convenience, as a function application is easy to integrate with AKV and AAD. This API implementation, being a somewhat ad hoc solution for the purposes of demonstrating the proof of concept implementation, warrants further consideration of security aspects in future development. For example, the API could also be implemented as a server on customer premises. Instead of AKV, another KMS could also be used.

An Azure function application allows each endpoint of the API to be implemented as a function in a supported programming language, which is executed each time the endpoint receives a request. JavaScript was chosen as the programming language for this implementation. The JavaScript runtime used by Azure to execute the functions is Node.js version 16.16.0 (LTS). Three libraries from the official SDK for Azure were used in the implementation. The library “@azure/keyvault-keys”, part of the official JavaScript SDK provided by Azure, was used to interact with AKV for key management [2]. Another library “@azure/secrets” is used to store backups of DEK as secrets on AKV [3]. The library “@azure/identity” was used to acquire credentials, that allow the function application to use the API of AKV [1].

6.2.1 Assigning Keys to Users

As required by the endpoints specified in the previous section, the API must map an RSA key pair to a user ID. The authenticated user can then call the endpoint via the web client to use their own private key. To achieve this, key pairs were created for each user of the service in AKV. Keys stored in AKV cannot be assigned to users directly, but they can hold arbitrary metadata in the form of tags. For each key pair of a user, a tag was assigned that denotes which user the key belongs to. See figure 6.1 for an example configuration.

Keys are queried from AKV by key name [4]. As an ad hoc solution to enable constant time queries for the user’s own key, each key was also named by the UUID of the owner user. The UUID is assigned by Azure automatically when the user account is created and stays constant. In production, it may be better to use a dedicated database for this purpose, so that keys can have arbitrary names.


Home > thesis-keys | Keys > 7bde40a5-25d9-46d8-9679-9231564b9c42 >
Save
Discard changes
Download public key

8e1dd415699745ddb0d0e0896ce21874
Key Version

Properties

Key type
RSA

RSA key size
4096

Created
1/19/2023, 12:22:13 PM

Updated
1/19/2023, 12:22:13 PM

Key Identifier
https://thesis-keys.vault.azure.net/keys/7bde40a5-25d...

Tags
8e1dd415699745ddb0d0e0896ce21874

Tag Name	Tag Value
user	Bob@thesistestorg.on...

Figure 6.1. A key pair assigned to a user on AKV.

6.2.2 Enabling Authentication and Authorization

To protect the private keys of users, the endpoints require that the caller has authenticated with AAD and has authorized the web client to use the API. The ID of the authenticated user is read by the function from the bearer token, and this ID is used to find the user's key. A function application can be configured to require authentication for all endpoints by adding an identity provider via the Azure web portal. Multiple providers are available, including proprietary providers and the protocol OpenID Connect. For this implementation, AAD was chosen as the provider. To enable a user of the web client to authenticate with AAD using PKCE, a SPA is configured as the authentication platform for the Azure application.

To authorize the Azure function application to use keys from AKV, it must be configured with the appropriate permissions. Each key operation type must be enabled explicitly to allow its use. [72] For this implementation, the “Get” and “List” permissions are required for key management. For cryptographic operations, the permissions “Sign” and “Decrypt” permissions are required. [2] Backups of DEKs are stored as secrets on AKV. Thus, to implement the endpoint to create DEK backups, the “Set” permission is required for secrets [3].

6.2.3 Example of an Endpoint Implementation

Code snippet 6.1 is an example of a function's code for signing a message as required by the endpoint specified in table 6.5. Less relevant parts of code were omitted, denoted by an ellipsis on a commented line. On line 12, the function code calls the helper function `getKeyOfCurrentUser` [2].

The helper function is shown in snippet 6.2. The ID of the user is used to find the key from AKV. The ID, in this case, is a UUID assigned to the user by Azure. It is read from a specific request header on line 4. A malicious client cannot spoof this header by inserting it themselves. The header is inserted by the Azure function application only after the access token of the request has been verified. Unauthorized requests will be denied and will not reach the function. [85] Reading the value of the header is therefore considered a secure way to deduce the ID of the calling user. The helper function returns the public key of the user by fetching it from AKV with the function `getKey` [2].

```javascript
const { DefaultAzureCredential } = require("@azure/identity");
const { KeyClient, CryptographyClient } = require("@azure/keyvault-keys");

module.exports = async function (context, req) {
  const credential = new DefaultAzureCredential();
  const client = new KeyClient(url, credential);
  const toSign = req.body["message"];

  // ...

  // Get the key of the current user calling this endpoint.
  const keyToUse = await getKeyOfCurrentUser(req, client);

  // Sign the message with the key of the current user.
  const crypto = new CryptographyClient(keyToUse.id, credential);
  const result = await crypto.signData("RS256", str2ab(toSign));

  // Encode the and return the result.
  context.res = {
    body: JSON.stringify({ result: b64String(result.result) }),
  };

  // ...
};
```

Snippet 6.1. *Azure function to sign a message with a private key.*

```javascript
function getKeyOfCurrentUser(req, keyClient) {
  return new Promise(async (resolve, reject) => {
    // Get the identity of the authenticated user.
    const authenticatedUserId = req.headers["x-ms-client-principal-id"];

    // ...

    // Get key by name and return.
    return keyClient.getKey(authenticatedUserId);
  });
}
```

Snippet 6.2. *A helper function to get the key of the current user from AKV.*

7. CLIENT-SIDE IMPLEMENTATION

For the client-side implementation of the MFCDS, MFW must provide functionality as required by the use cases described in section 5.3. The implementation of this functionality entails composing and parsing files of confidential documents. As specified by the use cases, some functionality of the system is provided by the key management API. The other functionality, such as encrypting and decrypting file content with a symmetric DEK, are provided by the client-side application. UI changes were also implemented to allow the viability of this system to be demonstrated in practice.

7.1 Using Cryptographic Algorithms on the Client-side

A JavaScript library was implemented that provides the required functionality for composing and parsing files of confidential documents. The library uses the Web Crypto API for running algorithms and importing and generating cryptographic keys [63].

7.1.1 Generating and Using a DEK

Creating and updating a document, as specified in sections 5.3.1 and 5.3.3, requires generating a symmetric DEK. The symmetric DEK can be generated via the function `generateKey` provided by the SubtleCrypto module. This function is given the key type and length in the first parameter, `"AES-GCM"` and `256` respectively. In the second parameter, are given the operations required of the key, in this case, `"encrypt"` and `"decrypt"`. The output of the function is a `CryptoKey` object, which can now be used for encryption and decryption. [63]

Web Crypto does not provide support for encrypting a stream of data. Data must be stored in a complete buffer before it is encrypted or decrypted.[63] Thus, when a DEK is used to encrypt a file, the entire file is first uploaded to the client-side application and then encrypted as one chunk. The downside is that the client-side application will consume more memory. This implementation could be improved by reading the file one small chunk at a time and encrypting each chunk with a different IV.

7.1.2 Importing Public RSA Keys

Public keys fetched from the key management API must be imported as `CryptoKey` objects, to enable their use for wrapping the DEK. To this end, the function `importKey` of the `SubtleCrypto` module is used. This function takes the desired RSA algorithm and hash algorithm as a parameter. The padding scheme, RSA-OAEP is used for wrapping the DEK. The hash algorithm is SHA-256. The imported public key can now be used to wrap DEKs. [63]

7.1.3 Wrapping a DEK

After encrypting the file, the DEK needs to be converted to a format where it can be wrapped by the public keys of users. The function `exportKey` of the `SubtleCrypto` module can be used. The generated DEK can be exported in the raw format, converting it to an array of bytes. The array can be directly encrypted by a public key, and then encoded as base64 to be stored in the metadata block. [63]

7.1.4 Verifying Signatures

The public key of the creator must be used to verify the signature of the metadata block when a confidential document is decrypted and viewed. The public key of the creator is requested via the API. Via the same `importKey` function, the creator's public key is imported. This time, the desired algorithm is RSASSA-PKCS1-v1_5, matching the signature algorithm of the API (see section 6.1.5). The imported RSA key is then converted to a `CryptoKey` object that can be used to verify signatures. The function `verify` of the `SubtleCrypto` module can then be used. [63]

7.1.5 Variant Using Elliptic-Curve Cryptography

Web Crypto supports ECDH. ECDH can be used to enable a similar implementation with ECC key pairs, as described in section 5.4.2. On the client side, the implemented logic differs from this implementation with RSA. The documentation of Web Crypto also provides an example of using ECDH to derive a shared AES key [63].

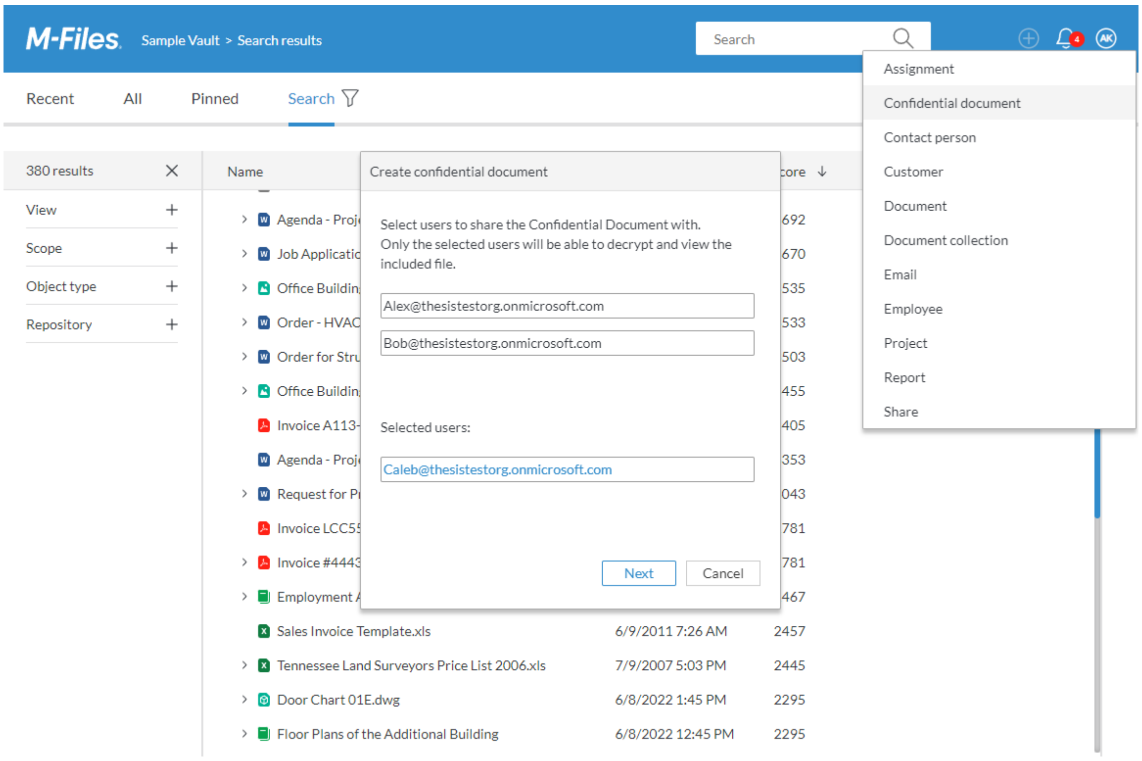
With ECDH, the shared key is derived between two parties, a sender and the recipient [44, p. 10]. Both derive the same key. However, with multiple recipients, each one will derive a different key. To share the same DEK key between all the recipients, for each one an AES KEK is derived. The shared DEK is then wrapped with the KEK of each recipient. To enable this implementation to use ECC, the function `deriveKey` from the `SubtleCrypto` module could be used to derive an AES KEK using ECDH. This derived AES KEK could then be used to wrap the DEK. Now, upon decrypting the file, the recipient would use an

ECC version of the API endpoint to unwrap the DEK (see section 6.1.4).

7.2 User Interface Implementation

To enable the use cases described in section 5.3, the implementation included additions to the UI of MFW. The UI implementation mainly consists of additions to the existing dialog system in MFW, which is implemented with the React UI component library [74].

The option to create a confidential document was added, according to the use case defined in section 5.3.1. Alongside other document types, users can now select the option to create a confidential document from the top bar of the UI. This opens a dialog for options specific to creating confidential documents. This is demonstrated in figure 7.1.



Create confidential document

Select users to share the Confidential Document with.
Only the selected users will be able to decrypt and view the included file.

Alex@thesistestorg.onmicrosoft.com

Bob@thesistestorg.onmicrosoft.com

Selected users:

Caleb@thesistestorg.onmicrosoft.com

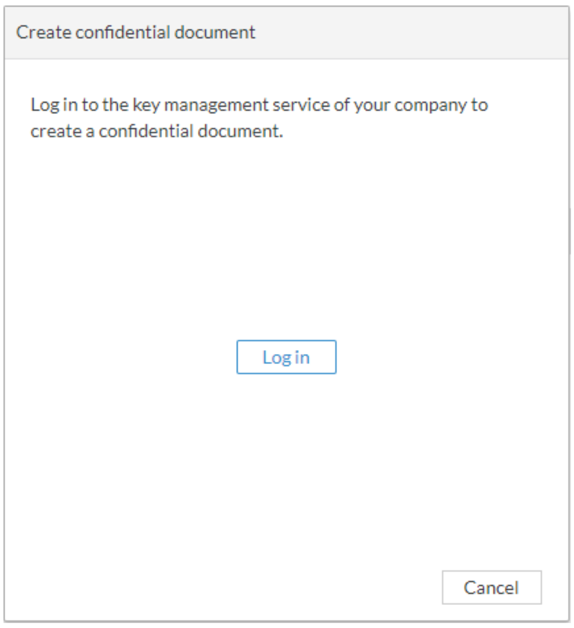
Sales Invoice Template.xls	6/9/2011 7:26 AM	2457
Tennessee Land Surveyors Price List 2006.xls	7/9/2007 5:03 PM	2445
Door Chart 01E.dwg	6/8/2022 1:45 PM	2295
Floor Plans of the Additional Building	6/8/2022 12:45 PM	2295

Figure 7.1. *An option to create a confidential document and the respective dialog window*

When opening the dialog, first the user is prompted to authenticate with the trusted identity management service, as depicted in figure 7.2. Consequently, the client is authorized to use the key management API. When the user clicks on the “Log in” button, the Microsoft login page is opened as a pop-up window, allowing the user to sign in with AAD.

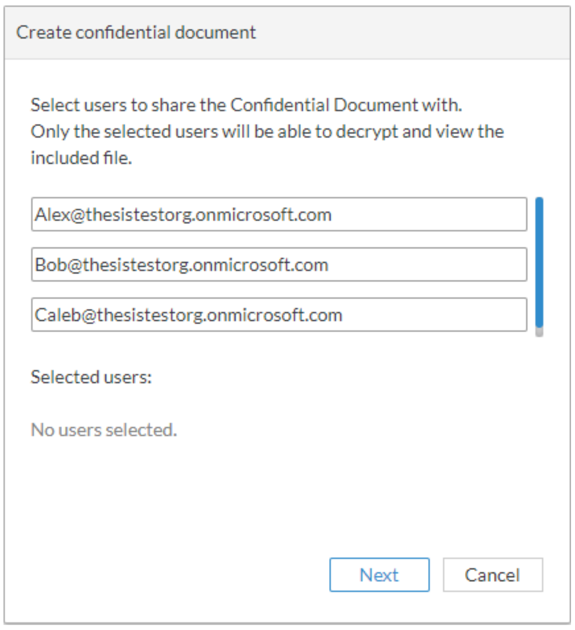
After the user has successfully signed in, the pop-up window is closed and the dialog displays the next page, shown in figure 7.3. On this page of the dialog, the user can select the allowed group of users by whom the document is to be accessed.

After the allowed group is selected, the last dialog page shown in figure 7.4 is displayed. Here the user can either drag and drop the file they wish to encrypt or click the prompt to


Create confidential document

Log in to the key management service of your company to create a confidential document.

Figure 7.2. *Prompt for the user to sign in*


Create confidential document

Select users to share the Confidential Document with. Only the selected users will be able to decrypt and view the included file.

Alex@thesistestorg.onmicrosoft.com

Bob@thesistestorg.onmicrosoft.com

Caleb@thesistestorg.onmicrosoft.com

Selected users:

No users selected.

Figure 7.3. *Dialog page for selecting the allowed group*

select one with the browser's file picker. After the file is uploaded, the user can encrypt it by clicking the "Encrypt" button. The file will then be encrypted, and only after that uploaded to MFS. Once uploaded, the user is shown the regular document creation dialog, where they can choose the class and other metadata of the confidential document.

This same dialog flow is also used for updating an already existing confidential document, according to the use case described in section 5.3.3. A confidential document can be updated like other documents in MFW, by dragging and dropping a file on the document in the UI listing.

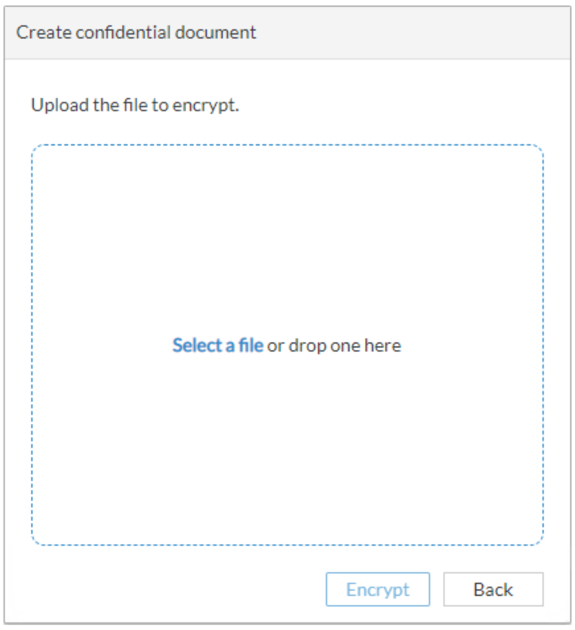

Create confidential document
Upload the file to encrypt.
Select a file or drop one here
Encrypt
Back

Figure 7.4. *Dialog page for uploading a file to be encrypted*

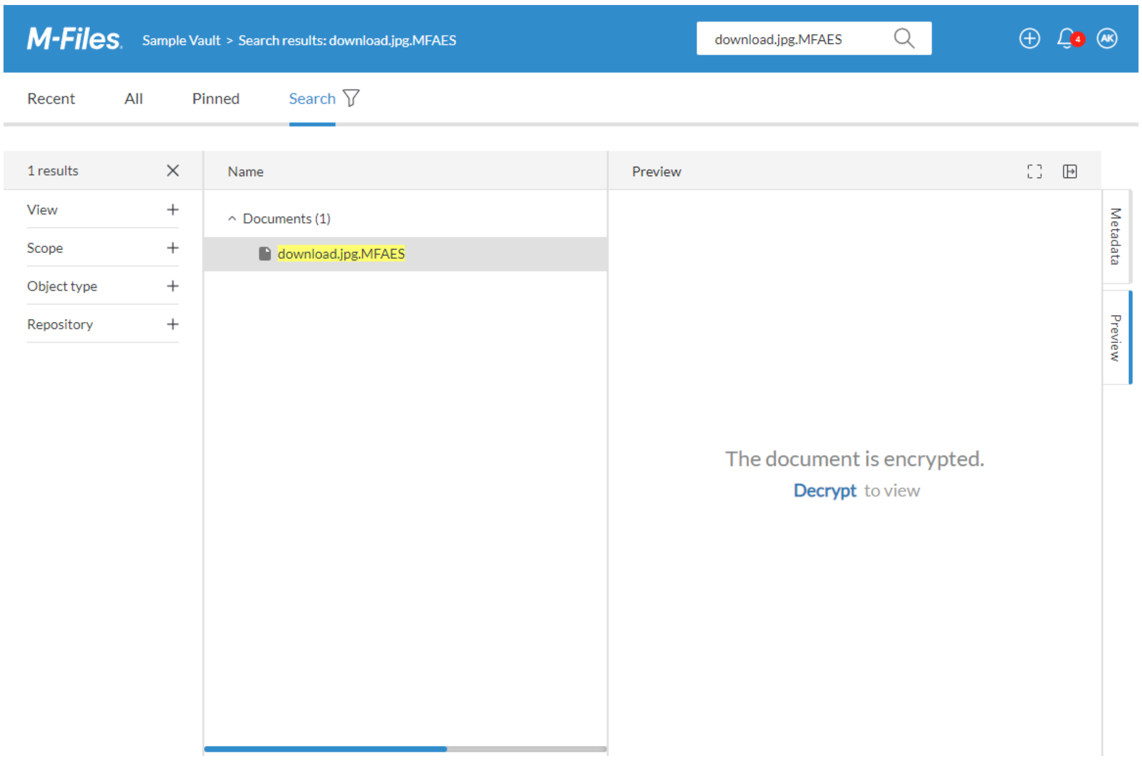


Figure 7.5. *Decrypt prompt in the preview pane*

Users in the allowed group of a confidential document must be able to decrypt the contained file and view it, as defined in section 5.3.2. To enable this, a decrypt prompt is shown in the preview pane whenever a confidential document is selected, as shown in figure 7.5. When a user clicks on the prompt to decrypt the file, the client begins the procedure to parse and decrypt it. If the user is not already signed in with the identity management service, they are shown the login page pop-up before proceeding. Then, if the user is in the allowed group, the file is decrypted and displayed in the preview pane. Otherwise, a toast notification is shown to indicate that the user is unauthorized. For any other error scenario while decrypting, such as a file with tampered content or an invalid signature, a toast notification is also shown.

8. EVALUATION OF PERFORMANCE

Performance tests were written to evaluate the viability of the MFCDS in terms of time efficiency. Tests were executed as scripts in the browser on Chromium version 108.0.5359.128. Tests were run on a computer with a GNU/Linux operating system, 16 gigabytes of DDR4 memory, and an Intel i5-8600K CPU. The tests measure the time performance of encrypting and decrypting a file in relation to control parameters: file size and the number of public keys that are included in the encrypted file. The tests measure both local execution time and time spent waiting for an API response for a call to the endpoint to sign data. It is important to separately measure the time waiting for a response, because network calls have inherent delay that is not related to the efficiency of the implementation.

Time measurements are recorded using the built-in Performance API provided by JavaScript runtimes [64]. For the encryption operation, time measurement begins when the client-side application starts composing the file by writing the metadata block and ends when the file is fully composed, with the encrypted content included. During this procedure, there is a call to the API to sign the metadata block, the duration of which is also measured. In the case of decryption, measurement begins when the client starts parsing the file and ends when the file is decrypted. There are no API calls during this procedure, so only the performance on the client side is measured. The cryptographic algorithms that are run during the measured operations are listed in table 8.1.

Function or endpoint	Algorithm	Key size (bits)	Type	Operation
SubtleCrypto.encrypt	AES-GCM 256 bit	256	Client	Encryption
SubtleCrypto.encrypt	RSA-OAEP	4096	Client	Encryption
API endpoint to sign	RSASSA-PKCS1-v1_5	4096	API	Encryption
SubtleCrypto.decrypt	AES-GCM 256 bit	256	Client	Decryption
SubtleCrypto.verify	RSASSA-PKCS1-v1_5	4096	Client	Decryption

Table 8.1. *The used cryptographic algorithms*

The performance of encryption and decryption operations was measured with respect to variable file size. The results are depicted in two charts in figures 8.1 and 8.2 respectively. Files of size 1, 250, 500, 750, and 1000 megabytes were used for testing. For each file size, the execution time was measured three times, the average of which is depicted in the charts. See appendix A and table A.1 for a complete table of measurements. From these charts, a linear trend can be observed for the execution time of both encryption and decryption. Predictably, as the size of the metadata block stays constant, the signing operation provided by the API is not affected by a change in file size, as can be observed in figure 8.1.

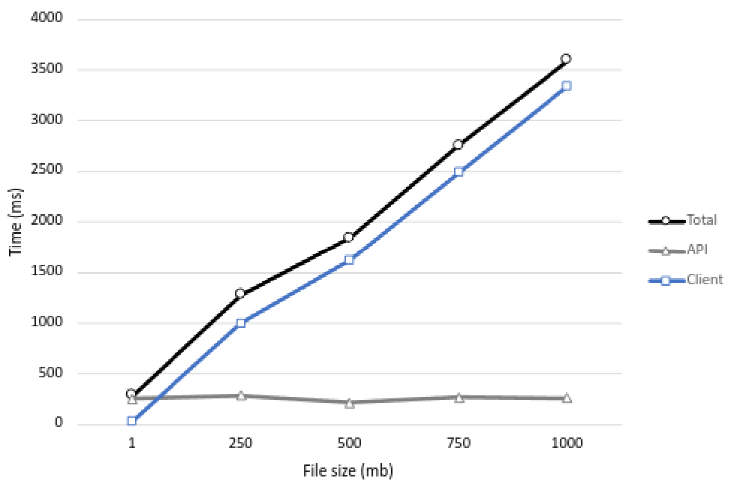


Figure 8.1. *Encryption peformance with variable file size*

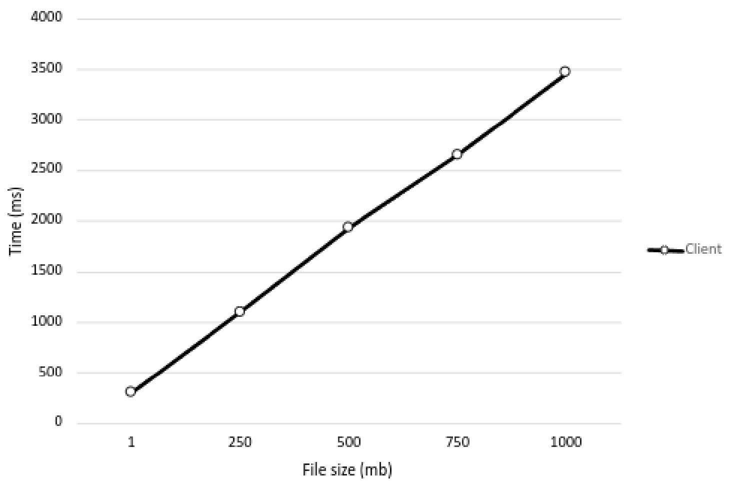


Figure 8.2. *Decryption peformance with variable file size*

The charts in figures 8.3 and 8.4 depict the measurement of encryption and decryption operations with respect to a variable number of public keys. A file size of 10 megabytes was used. The key amounts of 2, 100, 200, 300, and 400 were used. The same methodology applies as with the previous measurements. Complete measurements are available in appendix A in table A.2. The size of the metadata block increases with the number of public keys. Therefore, an increase in execution time can be observed for the API operation to sign the metadata block. A hardly noticeable increase can be observed for the client-side execution time of encryption, due to the plain text file size staying constant.

Some of the small increase can be caused by the reading and writing of the increasingly large metadata block. The decryption time, as can be observed from figure 8.4, does not appear to be impacted.

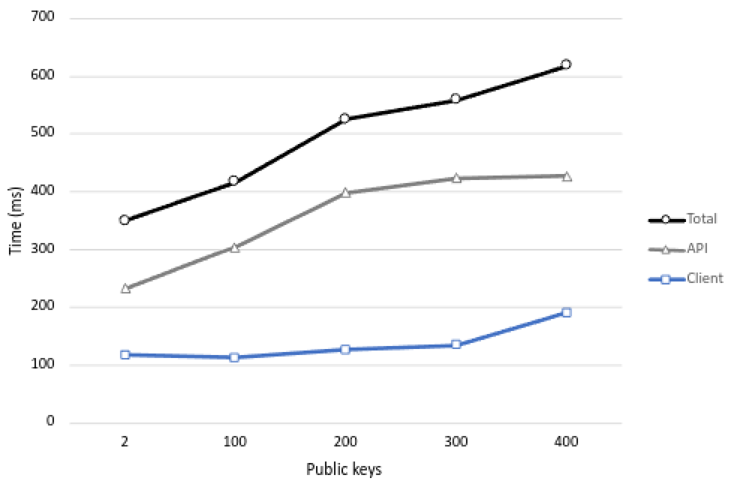


Figure 8.3. *Encryption peformance with variable number of public keys*

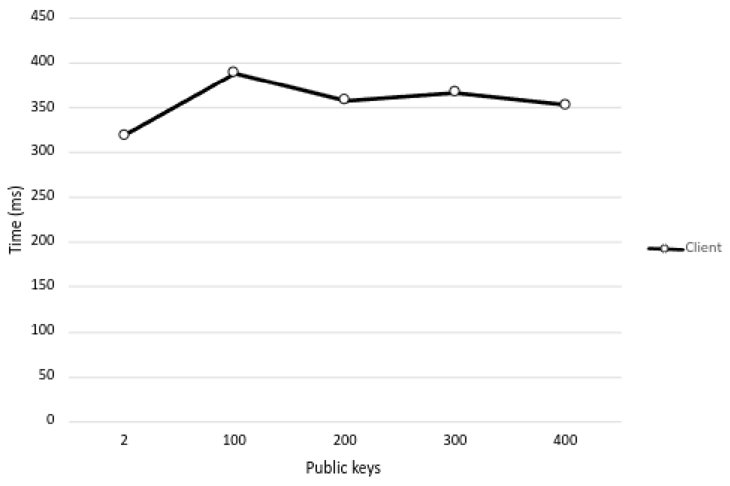


Figure 8.4. *Decryption peformance with variable number of public keys*

The performance of encryption and decryption appears sufficient for files in the measured size range. For much larger files, the delay caused may become unreasonable. The increase in execution time when the key amount increases is less drastic. Performance appears very good even for a fairly large number of public keys, staying well below one second for both encryption and decryption for the chosen file size. A combination of an increase in the number of keys and file size can be expected to be additive in nature (as opposed to multiplicative) because the delays in these measurements are caused in large part by independent factors. For example, the time to sign the metadata block is dependent on the number of included public keys. The signed metadata block does not increase in size as file size increases. Similarly, an increase in file size causes an increase in how long encryption or decryption with the symmetric DEK takes, but this is not affected by the number of public keys.

These measurements reinforce the conclusions of previous research on the efficiency of the Web Crypto API [62]. The functions provided by the Web Crypto API provided practically viable efficiency for this implementation.

9. CONCLUSION

This thesis investigated how client-side file encryption could be implemented for M-Files. The confidentiality of files was to be protected from M-Files itself, only allowing the intended users to access files that were encrypted on the client side. M-Files is used by enterprise customers, and as such the customer was to be enabled to centrally manage the keys of their users.

To meet the requirements, this thesis proposed the M-Files Confidential Document System (MFCDS). MFCDS is a client-side file encryption system, that enables the creation of confidential documents in M-Files. Via the web-client of M-Files, a user can now create a document in M-Files that is encrypted on their client and shared with a select group of other users. For managing user keys, a customer-owned KMS is integrated via a web API with a key management protocol. To the end-user a simple UI is displayed, while key management is handled automatically by client-side application logic. The implemented UI changes also enabled the practical demonstration of the MFCDS for basic use cases. The performance measurements of the implementation showed that the system is viable when used with files of moderate size. It can be expected that MFCDS can be used with the files of most documents, providing satisfactory efficiency for the end-user. Sharing the key to a confidential document, even with a large number of users, does not cause any significant performance impact. The system would be suitable even for customers with hundreds of users, where a large group of users might require access to the same document.

The implementation part of this thesis created a proof of concept for the MFCDS. The implementation can be further improved to make it viable for use in production. The design of the key management API can be revisited when there is a clearer picture of customer use-cases. For example, listing the keys of all users is an inefficient operation and could be improved by allowing the results to be filtered. The API endpoints could also be modified to take into account different key versions to support key lifecycles. The design of the API, and the protocol that it enables, while simplistic, was designed to be secure, following a similar design as key management protocols used elsewhere. However, this thesis did not address the implementation details of the API. The API implementation created here was a somewhat ad hoc solution for demonstration purposes. Further security analysis is warranted if the API were to be implemented for use in production. This thesis implemented MFCDS with support for RSA keys. Support could be added for ECC, and the details of how this could be done were documented in the applicable parts of the implementation chapters 6 and 7.

This thesis found that with an application of well-established cryptography, a client-side file encryption system can be built for an enterprise document management platform. The MFCDS appears viable from the standpoints of perceived security, usability, and performance. While there is much to improve, the fundamental concepts of the MFCDS should provide a solid base for future development. Actual productization can build on the efforts of this work.

REFERENCES

[1] *@azure/identity package. Azure documentation*. URL: https://learn.microsoft.com/en-us/javascript/api/@azure/identity (visited on 01/28/2023).

[2] *@azure/keyvault-keys package. Azure documentation*. URL: https://learn.microsoft.com/en-us/javascript/api/@azure/keyvault-keys (visited on 01/28/2023).

[3] *@azure/keyvault-secrets package. Azure documentation*. URL: https://learn.microsoft.com/en-us/javascript/api/@azure/keyvault-secrets (visited on 01/28/2023).

[4] *About Keys. Azure documentation*. 2022. URL: https://learn.microsoft.com/en-us/azure/key-vault/keys/about-keys (visited on 01/14/2023).

[5] Carlisle Adams and Steve Loyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. 2nd ed. Addison-Wesley, 2003.

[6] Salem T Argaw et al. "The state of research on cyberattacks against hospitals and available best practice recommendations: a scoping review". In: *BMC medical informatics and decision making* 19.1 (2019).

[7] Jean-Philippe Aumasson. *Serious Cryptography*. eng. No Starch Press, 2017.

[8] *AWS > Documentation > AWS KMNS > Developer Guide > AWS KMS concepts*. URL: https://docs.aws.amazon.com/kms/latest/developerguide/concepts.html (visited on 12/26/2022).

[9] *Azure Key Vault REST API reference. Azure documentation*. 2022. URL: https://docs.microsoft.com/en-us/rest/api/keyvault (visited on 01/14/2023).

[10] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. *MEGA: Malleable Encryption Goes Awry*. 2022. URL: https://mega-awry.io/pdf/mega-malleable-encryption-goes-awry.pdf (visited on 07/05/2022).

[11] E Barker et al. "SP 800-130. A Framework for Designing Cryptographic Key Management Systems, Draft Special Publication 800–130". In: (2010).

[12] Elaine Barker and Roginsky Allen. "SP 800-131A Rev. 2. Transitioning the Use of Cryptographic Algorithms and Key Lengths". In: *National Institute of Standards and Technology* (2019).

[13] Elaine Barker and William Barker. *SP 800-57 Part 2 Rev. 1. Recommendation for Key Management: Part 2 – Best Practices for Key Management Organizations*. 2019.

[14] Elaine Barker and Quynh Dang. *SP 800-57 Part 3 Rev. 1. Recommendation for Key Management: Part 3 – Application-Specific Key Management Guidance*. 2015.

[15] Elaine Barker et al. "SP 800-133 Rev. 2. Recommendation for Cryptographic Key Generation". In: (2020).

[16] Elaine Barker et al. *SP 800-57 Part 1 Rev. 5. Recommendation for key management: Part 1: General*. 2020.

[17] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. "Deterministic and Efficiently Searchable Encryption". In: Lecture Notes in Computer Science (2007).

[18] Mihir Bellare et al. "Deterministic Encryption: Definitional Equivalences and Constructions without Random Oracles". In: *Advances in Cryptology – CRYPTO 2008*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008.

[19] Tom Berson et al. "Cryptography as a Network Service". In: (2001).

[20] Moritz Borgmann et al. "On the security of cloud storage services". In: *Darmstadt (SIT Technical Report SIT-TR-2012-001)* (2012).

[21] Matthew Campagna and Shay Gueron. "Key Management Systems at the Cloud Scale". In: *Cryptography* 3.3 (2019).

[22] Daniele Catteddu. "Cloud Computing: Benefits, Risks and Recommendations for Information Security". In: *Web Application Security*. Communications in Computer and Information Science. Springer Berlin Heidelberg.

[23] Ramaswamy Chandramouli, Michaela Iorga, and Santosh Chokhani. "Cryptographic Key Management Issues and Challenges in Cloud Services". In: *Secure Cloud Computing*. 2013.

[24] Yan-Cheng Chang and Michael Mitzenmacher. "Privacy Preserving Keyword Searches on Remote Encrypted Data". In: *APPLIED CRYPTOGRAPHY AND NETWORK SECURITY, PROCEEDINGS*. Vol. 3531. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005.

[25] Long Cheng, Fang Liu, and Danfeng Daphne Yao. "Enterprise data breach: causes, challenges, prevention, and future directions: Enterprise data breach". In: *Wiley interdisciplinary reviews. Data mining and knowledge discovery* 7.5 (2017).

[26] *Client-side encryption for blobs. Azure documentation*. 2022. URL: https://docs.microsoft.com/en-us/azure/storage/blobs/client-side-encryption (visited on 07/14/2022).

[27] *Cloud Key Management Service > Documentation > Guides > Envelope Encryption*. 2022. URL: https://cloud.google.com/kms/docs/envelope-encryption (visited on 12/16/2022).

[28] Reza Curtmola et al. "Searchable symmetric encryption: improved definitions and efficient constructions". In: *Conference on Computer and Communications Security: Proceedings of the 13th ACM conference on Computer and communications security; 30 Oct.-03 Nov. 2006*. CCS '06. ACM, 2006.

[29] Josep Domingo-Ferrer et al. "Privacy-preserving cloud computing on sensitive data: A survey of methods, products and challenges". In: *Computer communications* 140-141 (2019).

[30] Morris Dworkin et al. *Advanced Encryption Standard (AES)*. 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197.

[31] Sascha Fahl et al. “Confidentiality as a Service – Usable Security for the Cloud”. In: *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2012.

[32] Ariel J Feldman et al. “{SPORC}: Group Collaboration using Untrusted Cloud Resources”. In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. 2010.

[33] Hal Finney et al. *OpenPGP Message Format*. RFC 4880. 2007. URL: https://www.rfc-editor.org/info/rfc4880.

[34] *FIPS 140-2. Security Requirements for Cryptographic Modules*. 2002.

[35] Craig Gentry. “Computing arbitrary functions of encrypted data”. In: *Communications of the ACM* 53.3 (2010).

[36] Eu-Jin Goh. *Secure Indexes*. Cryptology ePrint Archive, Paper 2003/216. 2003. URL: https://eprint.iacr.org/2003/216.

[37] *Google Workspace Admin Help > Security and data protection > Use client-side encryption for users’ data > About client-side encryption*. 2022. URL: https://support.google.com/a/answer/10741897?hl=en#zippy=%2coverview-of-cse-setup (visited on 11/06/2022).

[38] *Google Workspace for Developers > Client-side Encryption API > Reference > Google Workspace CSE API Reference*. 2022. URL: https://developers.google.com/workspace/cse/reference (visited on 12/16/2022).

[39] Harry Halpin. “The W3C web cryptography API: motivation and overview”. In: *Proceedings of the 23rd International Conference on world wide web*. WWW ’14 Companion. ACM, 2014.

[40] Dick Hardt. *RFC 6749: The OAuth 2.0 Authorization Framework*. 2012. URL: https://www.rfc-editor.org/rfc/rfc6749.

[41] Stephen Haunts. *Applied Cryptography in . NET and Azure Key Vault: A Practical Guide to Encryption in . NET and . NET Core*. eng. Apress L. P, 2019.

[42] Osama Hosam and Muhammad Hammad Ahmad. “Hybrid design for cloud data security using combination of AES, ECC and LSB steganography”. In: *International journal of computational science and engineering* 19.2 (2019).

[43] Md. Alam Hossain et al. “Measuring Interpretation and Evaluation of Client-side Encryption Tools in Cloud Computing”. In: *Security, Privacy and Reliability in Computer Communications and Networks*. 1st ed. Routledge, 2017.

[44] Kevin Igoe, David McGrew, and Margaret Salter. *Fundamental Elliptic Curve Cryptography Algorithms*. RFC 6090. 2011. URL: https://www.rfc-editor.org/info/rfc6090.

[45] Wayne Jansen and Timothy Grance. *SP 800-144. Guidelines on Security and Privacy in Public Cloud Computing*. 2011.

[46] Mika Javanainen. *What You Need to Know about Encryption in M-Files*. 2022. (Visited on 10/22/2022).

[47] Andrey Jivsov. *Elliptic Curve Cryptography (ECC) in OpenPGP*. RFC 6637. 2012. URL: https://www.rfc-editor.org/info/rfc6637.

[48] Yehuda Lindell Jonathan Katz. *Introduction to Modern Cryptography*. 2nd ed. Chapman and HallCRC, 2014.

[49] Michael Jones. *JSON Web Algorithms (JWA)*. RFC 7518. 2015. URL: https://www.rfc-editor.org/info/rfc7518.

[50] Michael B Jones. "RFC7517: JSON Web Key (JWK)". In: *Internet Engineering Task Force (IETF)* (2015).

[51] Kalyani Ganesh Kadam and Vaishali Khairnar. "Hybrid RSA-AES Encryption for Web Services". In: *International Journal of Technical Research and Applications", Special* 31 (2015).

[52] Burt Kaliski. *PKCS #7: Cryptographic Message Syntax Version 1.5*. RFC 2315. 1998. URL: https://www.rfc-editor.org/info/rfc2315.

[53] Emilia Käsper and Peter Schwabe. "Faster and timing-attack resistant AES-GCM". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2009.

[54] *Key Management Interoperability Protocol Specification Version 1.0*. 2015. URL: https://www.oasis-open.org/committees/download.php/37965/kmip-spec-1.0-cd-11.pdf (visited on 02/26/2023).

[55] Nasrin Khanezaei and Zurina Mohd Hanapi. "A framework based on RSA and AES encryption algorithms for cloud computing services". In: *2014 IEEE Conference on Systems, Process and Control (ICSPC 2014)*. IEEE, 2014.

[56] Jinyuan Li et al. *Secure Untrusted Data Repository (SUNDR)*. eng. 2003.

[57] *M-Files User Guide > Home > Daily Use > Managing Content > Editing Content > Version History*. URL: https://www.m-files.com/user-guide/latest/eng/History.html?hl=version%2Chistory (visited on 01/03/2023).

[58] *M-Files User Guide > Home > Introduction > M-Files Terminology*. URL: https://www.m-files.com/user-guide/latest/eng/M-Files_terminology.html (visited on 01/03/2023).

[59] *M-Files User Guide > Home > System Administration > Configuring M-Files > Customizing Server and Vault Behavior > Defining File Types for Indexing*. URL: https://www.m-files.com/user-guide/latest/eng/defining_files_for_indexing.html (visited on 01/03/2023).

[60] *M-Files User Guide > Home > System Administration > Setting Up and Maintaining M-Files > Managing Document Vaults > Vault Operations > Creating a New Document Vault > Document Vault Authentication*. URL: https://www.m-files.com/user-guide/latest/eng/document_vault_authentication.html (visited on 01/03/2023).

[61] *M-Files Web User Guide > Home > Overview*. URL: https://www.m-files.com/user-guide/web/latest/eng/web_overview.html (visited on 12/29/2022).

[62] Michael S MacFadden and Meikang Qiu. “Performance Impacts of JavaScript-Based Encryption of HTML5 Web Storage for Enhanced Privacy”. In: *2022 IEEE 7th International Conference on Smart Cloud (SmartCloud)*. IEEE. 2022.

[63] *MDN Web Docs > References > Web APIs > Crypto*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Crypto (visited on 11/05/2022).

[64] *MDN Web Docs > References > Web APIs > Performance*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance (visited on 02/04/2023).

[65] Peter Mell and Timothy Grance. *SP 800-146. The NIST Definition of Cloud Computing*. 2011.

[66] Silvio Micali, Charles Rackoff, and Bob Sloan. “The Notion of Security for Probabilistic Cryptosystems”. In: *SIAM journal on computing* 17 (1988).

[67] *OpenID Connect on the Microsoft identity platform. Azure documentation*. URL: https://learn.microsoft.com/en-us/azure/active-directory/develop/v2-protocols-oidc (visited on 01/03/2023).

[68] Alabi Orobosade et al. “Cloud Application Security using Hybrid Encryption”. In: *Communications on Applied Electronics* 7.33 (2020).

[69] Geong Poh et al. “Searchable Symmetric Encryption: Designs and Challenges”. In: *ACM computing surveys* 50.3 (2017).

[70] Raluca Popa et al. “CryptDB: protecting confidentiality with encrypted query processing”. In: *Proceedings of the Twenty-Third ACM Symposium on operating systems principles*. SOSP ’11. ACM, 2011.

[71] *Principles of Security and Trust Second International Conference, POST 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013, Proceedings. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage*. Springer Berlin Heidelberg, 2013.

[72] *Provide access to Key Vault keys, certificates, and secrets with an Azure role-based access control. Azure documentation*. 2022. URL: https://learn.microsoft.com/en-us/azure/key-vault/general/rbac-guide (visited on 01/28/2023).

[73] Hossein Rahmani et al. “Encryption as a Service (EaaS) as a Solution for Cryptography in Cloud”. In: *Procedia technology* 11 (2013).

[74] *React. A JavaScript library for building user interfaces*. URL: https://reactjs.org (visited on 01/05/2023).

[75] R Rivest, A Shamir, and L Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978).

[76] Nat Sakimura, John Bradley, and Naveen Agarwal. *RFC 7636: Proof Key for Code Exchange by OAuth Public Clients*. 2015. URL: https://www.rfc-editor.org/info/rfc7636.

[77] Abdul Salam. *Deploying and managing a cloud infrastructure : real world skills for the CompTIA cloud+ certification and beyond*. eng. John Wiley & Sons, 2015.

[78] *Security Guidance for Critical Areas of Focus in Cloud Computing v4.0*. 2017. URL: https://cloudsecurityalliance.org/artifacts/security-guidance-v4/ (visited on 07/08/2022).

[79] Adil Hussain Seh et al. "Healthcare Data Breaches: Insights and Implications". In: *Healthcare (Basel)* 8.2 (2020).

[80] *SpiderOak Support > One Backup > No Knowledge Explained*. 2022. URL: https://spideroak.support/hc/en-us/articles/115001855103-No-Knowledge-Explained (visited on 02/24/2023).

[81] Meltem Turan et al. *SP 800-132. Recommendation for Password-Based Key Derivation: Part 1: Storage Applications*. 2010.

[82] John R. Vacca. *Cloud Computing Security: Foundations and Challenges*. CRC Press, 2016.

[83] Mike Wills. *The Official (ISC)2 SSCP CBK Reference.* eng. 6th ed. John Wiley & Sons, Incorporated, 2022.

[84] Duane C. Wilson and Giuseppe Ateniese. ""To Share or not to Share" in Client-Side Encrypted Clouds". In: *Information Security*. Lecture Notes in Computer Science. Springer International Publishing, 2014.

[85] *Work with user identities in Azure App Service authentication. Azure documentation*. URL: https://learn.microsoft.com/en-us/azure/app-service/configure-authentication-user-identities (visited on 01/28/2023).

[86] Xun Yi. *Homomorphic encryption and applications*. SpringerBriefs in Computer Science. Springer, 2014.

APPENDIX A: PERFORMANCE MEASUREMENTS

Encryption			Decryption	
Size (MB)	Time total (ms)	Time API (ms)	Size (MB)	Time total (ms)
1	284	246	1	341
	262	238		305
	318	287		276
250	1542	296	250	1100
	1185	284		1071
	1110	272		1131
500	1823	223	500	1897
	1854	214		1969
	1852	219		1931
750	2727	247	750	2640
	2770	284		2671
	2764	273		2646
1000	3682	199	1000	3478
	3547	307		3448
	3566	271		3471

Table A.1. *Time measurements for encryption and decryption with variable file size*

Encryption			Decryption	
Public keys	Time total (ms)	Time API (ms)	Public keys	Time total (ms)
2	335	219	2	308
	387	273		339
	328	206		310
100	397	306	100	408
	401	297		389
	455	310		368
200	475	375	200	350
	528	419		366
	575	403		357
300	584	432	300	336
	548	422		440
	544	417		322
400	582	444	400	340
	606	430		366
	666	410		352

Table A.2. *Time measurements for encryption and decryption with a variable number of public keys*