B.Comp. Dissertation

Efficient Key Aggregation for Cloud Storage

By

Chong Yun Long

Department of Computer Science

School of Computing

National University of Singapore

2013/14

B.Comp. Dissertation

Efficient Key Aggregation for Cloud Storage

By

Chong Yun Long

Department of Computer Science

School of Computing

National University of Singapore

2013/14

Project No: H004900
Advisor: Assoc. Prof. Chang Ee-Chien
Deliverables:
Report: 1 Volume

Abstract

Data storage outsourcing to public cloud services is a growing trend among organizations. To preserve data confidentiality from potentially curious cloud providers, organizations may choose to employ client-side encryption for data stored in the cloud. The keys are kept by the organization and are issued as and when required. However, this gives rise to technical challenges, as key storage and distribution can incur substantial storage, bandwidth and computational costs. In this project, we examine these issues for time-series data, e.g. video surveillance data. To address these issues, various known cryptographic techniques such as GGM tree, key-aggregate cryptosystem (KAC), attribute-based encryption (ABE) and multi-dimensional range query over encrypted data (MRQED) are considered. We propose a key management scheme which requires $O(1)$ private storage overhead and $O(\log^2(n))$ contiguous key-generation time. Furthermore, the scheme supports the distribution of keys for varying video frame rates and video resolution. These objectives are achieved through the use of GGM-KAC hybrid structures, homomorphic signature schemes, as well as modifications to KAC.

Subject Descriptors:
E.3 Data Encryption

Keywords:
applied cryptography, range query, multimedia encryption, attribute-based encryption, key aggregation

Implementation Software and Hardware:
Macbook Air 2013 (Intel Core-i7-4650u), Mac OS v10.9.2, Python 2.7.5, Charm v0.43

Acknowledgement

I would like to thank my advisor Assoc. Prof. Chang Ee-Chien for his guidance and inputs throughout the project. I would also like to express my appreciation to Miss Zhu Xiaolu, Mr Zhang Chunwang, Mr. Fang Chengfang for providing me with valuable feedback and advice.

List of Figures

List of Tables

Table of Contents

Chapter 1

Introduction

With more and more organizations outsourcing data storage functions to cloud providers, the security and confidentiality of data in remote servers managed by third party have become a huge concern (Kamara & Lauter, 2010). Many providers such as Dropbox[1] and Google Cloud Storage[2] encrypt data using server-side encryption where keys are managed by cloud providers. As these service providers have possession of the keys, they can easily access any data stored on their platforms, potentially compromising data confidentiality. Moreover, since authentication and access controls are enforced by public servers, there is a possibility of unauthorized users gaining access to the data. Distribution of data across multiple servers for redundancy further exacerbates the risk of data leakage. In view of all these complications, there is a huge incentive for users to perform client-side encryption before uploading their data onto an external cloud server.

1.1 Problem and Motivation

Although client-side encryption presents a possible solution to some of the security problems associated with cloud storage, accessing and sharing of encrypted data comes with its own set of unique problems and challenges. We shall illustrate some of these problems with an example.

Alice decides to upload her personal video onto the cloud to free up valuable storage on her SSD. However, she does not trust the cloud server with her data for fears of possible data leakage, preferring to encrypt the video with her own key before uploading. To view the videos, Alice simply retrieves the video from the cloud and decrypts the video using her own key. Bob, a friend of Alice requests for permission to view the video. Alice grants Bob access to her video by sending her key to Bob via a secured channel. Bob downloads the encrypted video

[1] https://www.dropbox.com/

[2] https://cloud.google.com/products/cloud-storage

from the cloud and decrypts it using the keys he had gotten from Alice.

Suppose the length of the video is 50 minutes, and Alice wants to restrict Bob to only the last 20 minutes of the video. Hence, Bob must not be able to decrypt the first 30 minutes of the video. In this scenario, the previous scheme will not work if the entire video is encrypted with only 1 key. With just 1 key, Bob is able to decrypt the entire video, and Alice would have inadvertently granted Bob access to the first 30 minutes of the video. To overcome this problem, Alice encrypts each video frame with distinct keys and transfers the keys for the last 20 minutes of the video to Bob via a secured channel. Under this simple scheme, the number of keys Alice has to share with Bob is linearly proportional to the length of video. For a 20 min, 24 FPS video, Alice will have to share $24 \times 60 \times 20 = 28800$ keys with Bob. Such a large key size could pose a problem in a channel with limited network bandwidth.

There are a variety of techniques that can be employed to reduce the size of key which Alice has to send to Bob. Some of these techniques require extra storage, while some others require extra computational power. The above example has illustrated a few important considerations:

Bandwidth The total keysize that Alice needs to transfer to Bob should be small.

Storage The total size of encryption keys that Alice has to store should be small.

Runtime Bob should not take a long time to decrypt the video frames.

Principle of least privilege Bob only has access to the last 20 minutes of the video. Hence, the first 30 minutes of the video must not be revealed to Bob.

Collusion-resistant Suppose Charles is also granted decryption keys to the same video. Bob and Charles should not be able to collude to decrypt video frames that neither of them can decrypt.

Expressiveness Alice should have the flexibility to restrict Bob's access to the video to either a single frame or a set of frames. Similarly, Bob should have the flexibility to request for decryption keys to either a single frame or a set of frames.

1.2 Objective

Alice faces a problem of large key size each time she shares keys with Bob. To solve Alice's problem, we want to devise a cryptographic and key management scheme that reduces the keysize needed for sharing of encrypted data. In this project, we will be focusing our efforts

on multimedia data. The proposed scheme should also balance size of keys transferred with encryption/decryption time and storage overhead. Bob should not need to wait for hours to decrypt the video he got from Alice. Also, the ciphertext produced should not be significantly larger than plaintext.

1.3 Contributions

We have made the following contributions in this project:

- Explored several existing techniques such as GGM construction, attribute-based encryption (ABE), Multi-dimensional Range Query over Encrypted Data (MRQED) and key-aggregate cryptosystem (KAC). We have also built prototypes to compare the empirical performance of ABE and KAC.
- Developed a hybrid KAC-GGM scheme that is able to support contiguous decryption in $\log^2(n)$ time.
- Observed that the runtime complexity of KAC decryption can be reduced from quadratic time to linear time for arithmetic sequences through the reuse of computed intermediate values.
- Developed a homomorphic MAC scheme that reduces storage requirements of key granter from $\Theta(n)$ to $O(1)$.
- Developed an encryption scheme which makes use of symmetric encryption to support fast decryption of layered content.

Chapter 2

Related work

In this chapter, we examine related cryptographic schemes and evaluate how these schemes could be used as a possible solution to achieve our objectives. In Section 2.1, we will explore the use of GGM trees as a possible solution to reduce key size. Section 2.2 introduces ABE, a system which makes use of user attributes and access policies to manage ciphertext and keys. In Section 2.3, we evaluate MRQED, a scheme that is used for encryption of multi-dimensional data. Finally, we present KAC in Section 2.4, a cryptosystem that is capable to aggregate any set of keys into a constant-size aggregate key. Many of these schemes are based on the use of bilinear pairings, a cryptographic operation which will be discussed in Section 2.2.5.

2.1 GGM Tree

GGM Trees (Goldreich, Goldwasser, & Micali, 1986) are used to generate pseudorandom numbers, or in our case encryption/decryption keys.

2.1.1 GGM Construction

Let G be a cryptographically strong pseudorandom bit generator (CSB) that stretches a s-bits long input into a $2s$-bits long output. Denote G_0 to be the first k bits and G_1 to be the last k bits of the output from G.

$$G : \{0,1\}^s \rightarrow \{0,1\}^{2s}, G(x) = G_0(x) \,||\, G_1(x)$$

Given seed x, we are able to construct a GGM tree with the root node, left child and right child represented by $x, G_0(x), G_1(x)$ respectively. Fig. 2.1 shows a GGM tree with 4 leaf nodes. The security property of the CSB ensures that it is difficult to derive the right subtree using the left subtree and vice-versa.

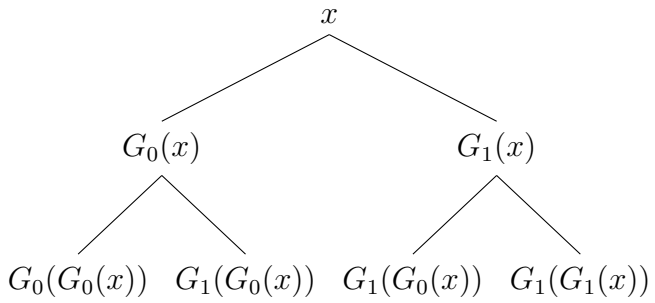


Figure 2.1: GGM tree

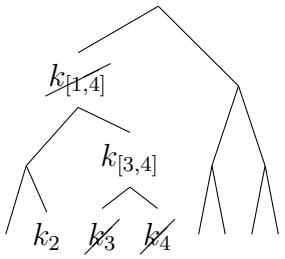


Figure 2.2: Releasing keys for $[2, 4]$

The leaf nodes of the GGM tree can be used as encryption keys for the video frames (frame keys). To release the frame keys to Bob, Alice can aggregate the leaf nodes under their parent nodes, reducing the number of nodes Alice needs to transfer to Bob. We use $[x, y]$ to denote frames x to y inclusive.[1]

In Fig. 2.2, Alice just needs to send nodes $\{k_2, k_{[3,4]}\}$ for Bob to obtain the keys for $[2, 4]$. However, Alice cannot send Bob $k_{[1,4]}$ because Bob is able to derive k_1, thus allowing him to decrypt frame 1 which he is not entitled to have access to. Alternatively, Alice could also choose to send $\{k_2, k_3, k_4\}$ to Bob. However, this is less efficient than sending Bob $\{k_2, k_{[3,4]}\}$.

2.1.2 Evaluation

Alice only have to send Bob a logarithmic-size key for contiguous range queries[2]. However, non-contiguous queries can yield a worst-case keysize of $O(n)$ where n is the total number of frames in the video. This can occurs if the requested frame keys follows an arithmetic sequence, $a_i = a_1 + (i - 1)d$, where $d \geq 0$ and i being the frame number ($i \geq 1$). In this situation, none of the frame keys requested can be aggregated under a common parent node. For example, consider the case where Bob request for keys to all the odd frames. Consequently, the requested frames can be expressed as an arithmetic sequence of the form $a_i = 1 + 2(i - 1)$. Since none of the frames can be aggregated, Alice needs to send $n/2$ leaf nodes to Bob.

Alice does not have to store all of the frame keys as she can derive the frame keys from the root node, albeit at the cost of increased computational time. As a result, storage overhead is reduced from $\Theta(n)$ to $O(1)$.

2.2 Attribute-Based Encryption

Attribute-Based Encryption (ABE) (Bethencourt, Sahai, & Waters, 2007; Waters, 2011; Goyal, Pandey, Sahai, & Waters, 2006; Sahai & Waters, 2005; Lewko, Sahai, & Waters, 2010) is a

[1] $[x, y] = \{x, x + 1, ..., y\}$
[2] Queries in the form $[x, y]$

scheme where the ciphertext and the corresponding keys are dependent on attributes and access structures. Depending on the type of access policy, ABE can be classified broadly into two catgories, namely cipertext-policy ABE (Bethencourt et al., 2007; Waters, 2011) and key-policy ABE (Goyal et al., 2006; Lewko et al., 2010).

2.2.1 Ciphertext-Policy ABE

We introduce the notion of ciphertext-policy ABE (CP-ABE) with a simple example. Alice wants to share a video with her mother, father, brother, her classmates of class 4C (year 4, class C) as well as her female choir friends. In CP-ABE, each key that a user possesses will be associated with a set of descriptive attributes. The attributes involved in this example are `mother`, `father`, `brother`, `year 4`, `class C`, `choir`, `female` and `male`. Alice's mother's key will be encrypted with the attribute set $\{$`mother, female`$\}$, represented by $K_{\{\texttt{mother, female}\}}$.

The criterion needed to decrypt the video is termed as an "access policy". We represent access policy with a boolean formula involving **AND** and **OR** gates. In this example, Alice is able to specify the video's access structure as: (`mother` **OR** `father` **OR** `brother` **OR** (`year 4` **AND** `class C`) **OR** (`female` **AND** `choir`)). The boolean formula is then transformed into a policy tree as shown in Fig. 2.3. The attributes form the leaves of the access tree while the **AND** and **OR** gates form the inner nodes.

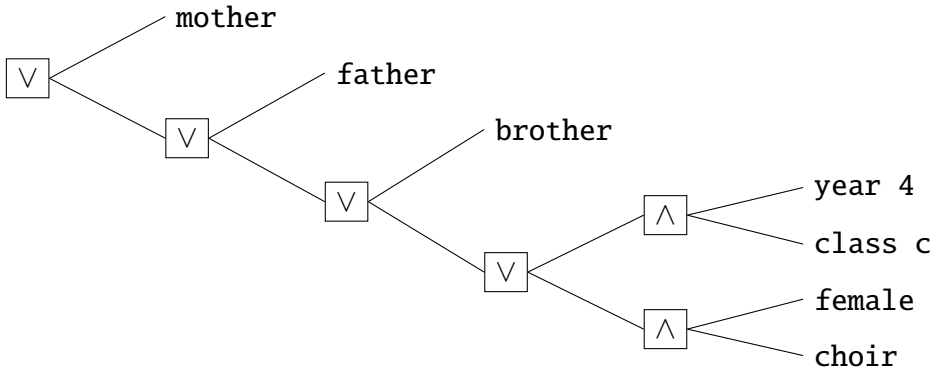


Figure 2.3: Access Tree with **AND**, **OR** gates

As a generalization of ABE, we are able to replace **AND** and **OR** nodes with threshold gates shown in Fig. 2.4. For instance, a 2/5 threshold gate is evaluated to true only if any 2 out of 5 subtrees of the threshold gate are evaluated to be true. As a generalization, an **AND** gate can be represented by a 2/2 gate and an **OR** gate can be represented by a 1/2 gate.

Using the same example, we consider Bob who is in the same year and same choir as Alice and Charles who is Alice's classmate. Correspondingly, Bob receives $K_{\{\text{year 4, male, choir}\}}$ while Charles receives $K_{\{\text{year 4, class C, male}\}}$. Bob is unable to decrypt the video because he neither satis-

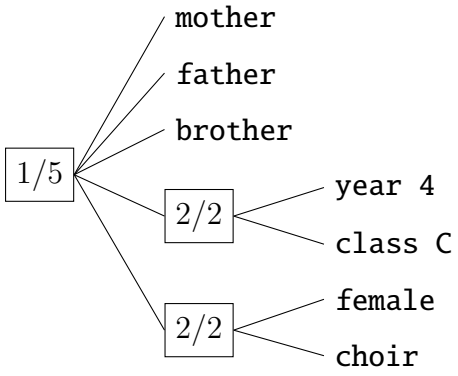


Figure 2.4: Access Tree with threshold gates

fies the female[3] choir member criteria nor the class c criteria. In contrast, the access tree evaluates to be true with respect to attribute set $\{\texttt{year 4}, \texttt{class C}\}$. Since, $\{\texttt{year 4}, \texttt{class C}\} \subset \{\texttt{year 4, class C, male}\}$, Charles in possession of $K_{\{\text{year 4, class C, male}\}}$ is able to decrypt the video. In CP-ABE, the access policy is associated with ciphertext and the attributes are associated with the keys, hence the name "ciphertext-policy".

2.2.2 Key-Policy ABE

In KP-ABE, the access policy is associated with keys and the attributes are associated with the ciphertext, hence the name "key-policy". We reference to the example discussed in Section 1.1. Instead of sending a distinct key for each frame, Alice uses KP-ABE to encrypt each video frame with the frame number as its attribute. For a video with n frames, Alice can issue to Bob $K_{\{(n-28799)\vee(n-28798)\vee\cdots\vee n)\}}$. The access policy in this case is a series of **OR** gates for every frame in the last 20 minutes of the video. Bob will thus be able to decrypt all frames in the range $[n - 28799, n]$. However, the key transmitted is still linear in size. An algorithm that represents contiguous sets more efficiently[4] will be introduced in Section 2.2.7.

2.2.3 Realization of ABE

In this section, we describe the algorithms used in KP-ABE. However, we will not discuss the details of KP-ABE construction in this report. The KP-ABE scheme in (Lewko et al., 2010) consists of 5 algorithms, as shown in Table 2.1.

[3]The `male` attribute is not represented in the access tree

[4]It is possible to represent $[n - 28799, n]$ using logarithmic number of attributes

Algorithm	Inputs	Outputs
Setup	Generators g, h Exponents $a', a'', b \in \mathbb{Z}_p$	Master Key (MK) Public Key (PK)
Encryption	PK, Message (M), Attribute set (γ)	Ciphertext (E)
Key Generation	MK, PK, Access structure ($\mathbb{A}$)	Private Key (D)
Decryption	E, D	M

Table 2.1: KP-ABE algorithms

It is easy to see how this scheme can be adapted to a cloud storage. Alice runs **Setup** and uses **Key Generation** together with access structure $\mathbb{A}$ specified for Bob, producing Bob's private key (D). Alice next encrypts M (video frame) under attribute set γ (frame number) and stores it on the cloud. Bob is then able to use D to decrypt M if γ satisfies $\mathbb{A}$. Also, **Key Generation** randomizes the keys generated to prevent collusion, a situation where two different parties are able to combine their keys to decrypt messages that neither of them has access to.

2.2.4 ABE Operations and Structures

In this section, we take a look at some of the operations and structures that are used in ABE.

2.2.5 Bilinear Map

Bilinear Map is a fundamental cryptographic operation used in ABE as well as many other encryption schemes. Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3$ be groups of prime order p, $u \in \mathbb{G}_1, v \in \mathbb{G}_2$. A bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$ has the following properties:

Bilinearity $a, b \in \mathbb{Z}_p$, $e(u^a, v^b) = e(u, v)^{ab}$

Non-degeneracy $e(u, v) \neq 1$

Computable e can be computed efficiently for practical purposes.

2.2.6 Monotonic Access Schemes

Most ABE schemes are based on monotonic access schemes. Let $\mathbb{A} \subseteq 2^{\{P_1, P_2, \ldots, P_n\}}$, where $\{P_1, P_2, \ldots, P_n\}$ refer to the set of parties. A collection $\mathbb{A}$ is monotonic if $\forall B, C$, if $B \in \mathbb{A}$ and $B \subseteq C$ then $C \in \mathbb{A}$. An access structure is a collection $\mathbb{A}$ of non-empty subsets, that is $\mathbb{A} \subseteq 2^{\{P_1, P_2, \ldots, P_n\}} \setminus \{\emptyset\}$. This means that we do not consider the unauthorized sets which are not in $\mathbb{A}$ An example of a non-monotonic policy would be "students who do not belong to Alice's class".

One possible solution would be to represent negation as a separate attribute but this would result in a huge increase in the number of attributes required. There are however ABE schemes proposed which allow policies to be expressed in non-monotonic access structures (Ostrovsky, Sahai, & Waters, 2007). For this project, we will not be considering non-monotonic access structures.

2.2.7 Range Comparison

Using "bag of bits" representation (Bethencourt et al., 2007), we are able to support integer comparisons using $O(n)$ gates, for an n bit integer. We represent 4 bit attribute such as "$x = 9$' using 4 attributes {`x:1***`, `x:*0**`, `x:**0*`, `x:***1`}. **AND**, **OR** gates is then used to implement comparisons such as "$x < 10$" with a policy illustrated in Fig. 2.5.

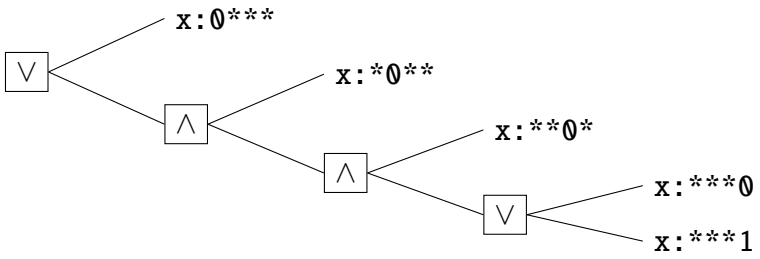


Figure 2.5: Policy tree for $x < 10$

It is easy to verify that "$x = 9$" satisfies the policy tree. 4 bit integers which are larger than or equal to 10 such as "$x = 13$" represented as {`x:1***`, `x:*1**`, `x:**0*`, `x:***1`}, is unable to satisfy the policy tree.

2.2.8 Evaluation

Since ABE's key and ciphertext size grow linearly with respect to the number of attributes, the "bag of bits" representation will result in a key and ciphertext size logarithmic to the value of the numerical attribute. Using KP-ABE, Alice would encrypt each video frame together with their respective frame number using the "bag of bits" representation. Alice then generates Bob's key restricting him to a specific range of frames. This is done again using the "bag of bits" representation. Using this method, Alice only has to send one single logarithmic-size key to Bob, resulting in a huge reduction in key size.

Depending on the number of attributes and type of access policies, ABE can be computationally expensive due to the pairing and exponentiation operations it has to perform. More investigations are needed to determine its efficiency for large datasets.

2.2.9 Multi-message Ciphertext Policy ABE

Multi-message Ciphertext Policy ABE or MCP-ABE (Wu, Wei, & Deng, 2013) is a scheme that is built upon CP-ABE, catered specifically for scalable media. The paper proposed segmentation of media content into l layers, each with different privilege rights. A video file for instance can be segmented into subtitles, audio, low quality video (LQ) and high quality video enhancement (HQ). Each of this layer is associated with an access privilege: $p_l, p_{l-1}, \ldots, p_1$, and p_{i+1} having a higher privilege than p_i. The highest privilege level is p_l while the lowest is p_1. Suppose that Alice has access to HQ enhance layer with the highest privilege p_l. Alice should also have access to all other layers of lower privileges (subtitles, audio, low quality video). Hence, p_l should grant access to the keys of not only k_l but also all the other keys. This is accomplished through the use of a chain hash.

$$k_j = H(k_{j+1}||j), j = l-1, \ldots, 2, 1 \tag{2.1}$$

k_l is hashed repeatedly to generate keys for all other layers. It is not computationally feasible to generate k_{j+1} from k_j due to preimage resistance of the cyptographic hash function.

Unlike in CP-ABE where each access structure is associated with a single ciphertext message, MCP-ABE associates multiple messages with a single policy tree as shown in Fig. 2.6. Each message is a symmetric key which is used for decryption of the respective layers. The keys are derived using the hash chain shown in Equation 2.1. The ability to associate encrypt multiple messages using a single access tree allows us to have a hierarchy of access levels.

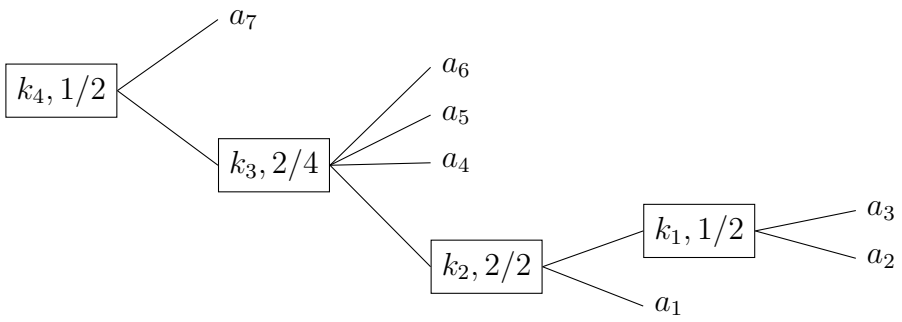


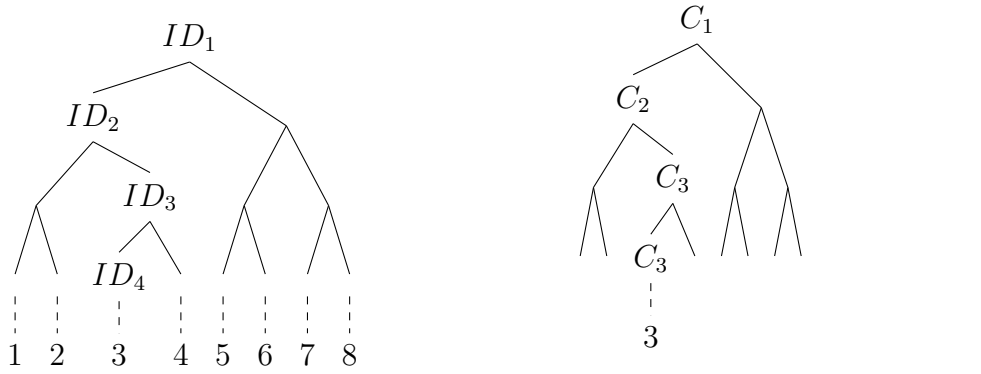
Figure 2.6: MCP-ABE access tree

Each threshold gate is associated with a particular message, which in this case is a symmetric key used for media decryption. Since the keys are generated via a hash chain, having a_7 would allow the user to calculate k_4. Using k_4, a user would be able to use Equation 2.1 to obtain all other keys (k_3, k_2, k_1).

2.3 Multi-dimensional Range Query over Encrypted Data

Suppose Alice organizes the frames belonging to different videos into a multi-dimensional scheme by tagging them with GPS latitude and longitudinal coordinates (la, lo), timestamp (t) of the video, picture resolution (p) and frame number (f). Each video frame can be referenced by tuple (la, lo, t, q, f). Multi-dimensional Range Query over Encrypted Data (Shi, Bethencourt, Chan, Song, & Perrig, 2007; Bethencourt, h. Hubert Chan, Perrig, Shi, & Song, 2006), or MRQED is a scheme that allows for small key sizes in the encryption and decryption of such multi-dimensional data.

MRQED organizes the public keys and private keys (ID) into interval trees, each node in charge of encrypting a particular range/point.



(a) Encryption for frame 3 under 4 different ID

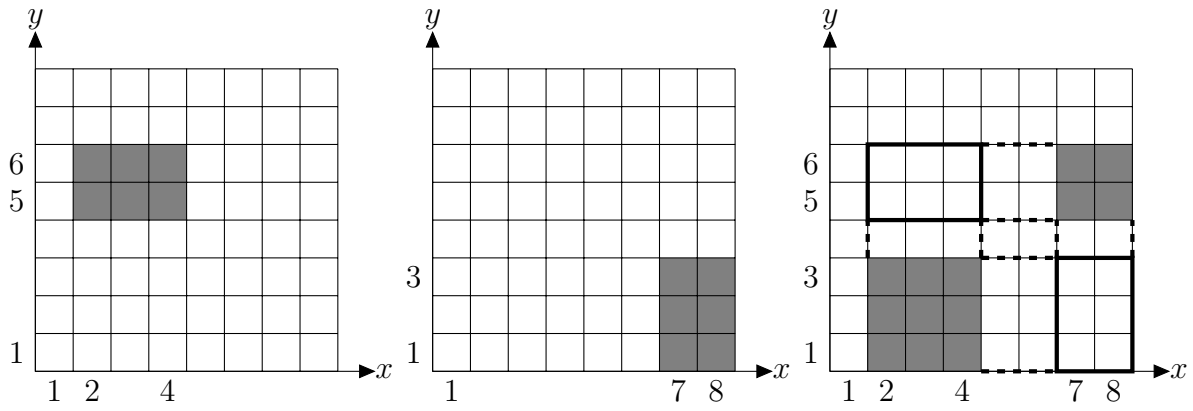
(b) 4 different ciphertexts for frame 3

Figure 2.7: Encryption under MRQED

In Fig. 2.7a, Alice has to encrypt frame 3 a total of 4 times under $\{ID_1, ID_2, ID_3, ID_4\}$ since all of these nodes covers frame 3. As a result, 4 different ciphertexts will be produced for f_3 as shown in Fig. 2.7b. Encryption of a single frame will take $O(\log(n))$ time. Each piece of ciphertext (a single encrypted frame) will occupy $O(\log(n))$ space. Keys are released in a manner similar to the GGM tree illustrated in Fig. 2.2.

2.3.1 MRQED for multiple dimensions

MRQED can be extended to multiple dimensions. Intuitively, we build D trees for D dimensions. In Fig. 2.8a, Alice wants to release keys for $x1 = [2, 4], y1 = [5, 6]$ to Bob. To do so, we find a minimum cover for x_1 and release key set k_{x1}. Likewise for $y1$, we obtain the minimum cover key set k_{y1}. To decrypt hyperrange $\{x1, y1\}$ we send Bob k_{x1} and k_{y1}. This is a very efficient construction, with a decryption key size of only $O(D \log(n))$ with no additional overhead.



(a) $x = [2, 4], y = [5, 6]$ (b) $x = [7, 8], y = [1, 3]$ (c) $x = [2, 4], y = [1, 3]$ and $x = [7, 8], y = [6, 5]$ after collusion

Figure 2.8: Collusion Problem

However, this naive construction introduces the problem of collusion between different parties. Suppose Charlie obtains keys for the region $x2 = [7, 8], y2 = [1, 3]$, shown in Fig. 2.8b. Bob is able to collude with Charlie to obtain the following additional keys for the regions $\{x1, y2\}$ and $\{x2, y1\}$, as shown in Fig. 2.8c.

Alternatively, we could trivially take the cross product of the different paths to the roots of each dimension. However, this results in a huge expansion in key size and ciphertext size since we now have $\prod^{D} O(\log(n)) = O(\log(n))^D$ ciphertext size per frame, where D is the number of dimensions.

MRQED (Shi et al., 2007) presents a method that prevents the collusion attack without incurring an increase in key and ciphertext size. Instead of issuing $\{k_x, k_y\}$, random variables are added to the key, producing $\{u_x k_x, u_y K_y\}$, such that $u_x u_y = i$, where i is an invariant to keep the scheme consistent. Using the same example, Alice would have $\{\tilde{u}_x k_{x1}, \tilde{u}_y k_{y1}\}$ and Bob will have $\{\tilde{u}'_x k_{x2}, \tilde{u}'_y k_{y2}\}$. Since $\tilde{u}_x \tilde{u}'_y \neq i$ and $\tilde{u}'_x \tilde{u}_y \neq i$, Bob and Charlie are unable to collude with each other to obtain additional key information.

2.3.2 Evaluation

Both MRQED and ABE have similar space requirements, making heavy usage of tree structures. Moreover, MRQED scheme requires reorganization of data into a rigid predefined structure and is better employed for multi-dimensional data. On the other hand, ABE does not have such a limitation, as we are able to associate arbitrary names and values to the keys and ciphertexts, providing much more flexibility than MRQED.

2.4 Key-Aggregate Cryptosystem

Key-aggregate cryptosystem, or KAC (Chu, Chow, Tzeng, Zhou, & Deng, 2013) is a scheme that allows use of a constant-size decryption key to decrypt any subset of ciphertexts.

In this scheme, messages are encrypted under predefined ciphertext classes, and are identified by an index. To share keys belonging to ciphertext classes $1, 3, 6$, Alice will have to send $\{k_1, k_3, k_6\}$ to Bob, which is not efficient.

KAC provides a mechanism to aggregate the keys into a single key called an aggregate key (K_S) which is constant-size. For this example, Alice aggregates k_1, k_3, k_6 into a constant-size aggregate key K_S, where $S = \{1, 3, 6\}$. She then forwards K_S along with a public parameter (`param`) to Bob. Bob is able use K_S together with `param` to decrypt ciphertext classes belonging to either 1, 3 or 6. K_S is constructed to be collusion-resistant.

2.4.1 Realization of KAC

KAC provides the power to decrypt messages belong to a set of ciphertext classes using only a constant-size key. The 5 algorithms of KAC is shown in Table 2.2. We will not go into the details of the construction of KAC in this section.

Algorithm	**Inputs**	**Outputs**
Setup	Security parameter (1^λ), number of ciphertext classes (n)	Public parameter (`param` $= \langle g, g_1, \ldots, g_n, g_{n+2}, \ldots, g_{2n} \rangle$)
KeyGen	`param`	Public key (`pk` $= v = g^\gamma$), Master secret key (`msk` $= \gamma$)
Encrypt	`pk`, index $1 \leq i \leq n$ denoting ciphertext class, massage (`msg`), `param`	ciphertext ($\mathfrak{C} = \langle c_1, c_2, c_3 \rangle$)
Extract	`msk`, set of ciphertext classes (S), `param`	Aggregate key ($K_S = \prod_{j \in S} g^\gamma_{n+1-j}$)
Decrypt	K_S, S, i, $\mathfrak{C}$, `param`	`msg` if $i \in S$

Table 2.2: KAC algorithms

Alice uses `pk` generated in **KeyGen** stage along with ciphertext class i to encrypt `msg`, producing $\mathfrak{C}$. Bob with access rights to set S of ciphertext classes is entrusted with K_S. K_S can be used to decrypt $\mathfrak{C}$ if $i \in S$. It is theoretically impossible for decryption keys to be constant in size for all possible combinations of ciphertext classes. The constant-size aggregate key is archived by encapsulating and delegating the information needed for encryption and decryption to `param`, which is $\Theta(n)$ in size.

Although the use of K_S together with `param` is not of $O(1)$ in size. However, such a scheme can be very beneficial in the context of unsecured cloud storage. For instance, Alice stores $\mathfrak{C}$

and param in the cloud and sends K_S to Bob via a private channel. In this case, the small size of K_S in this case is desirable because the private channel may be limited in bandwidth. Thus, it can be seen that KAC is an excellent scheme for handling time-series data, as data points can be represented as distinct ciphertext classes.

2.4.2 Key Extension

Since the number of ciphertext classes n has to be be defined during **Setup**, extending the ciphertext classes using existing msk and pk can be difficult. In (Chu et al., 2013), an extension scheme is proposed with l (v, γ) key pairs, supporting up to $O(ln)$ ciphertext classes. Each class is now represented by a 2-dimensional coordinate $\{(i, j)|1 \leq i \leq l, 1 \leq j \leq n\}$, where $\mathtt{pk}_l = \{v_1, \ldots v_l\}, \mathtt{msk} = \{\gamma_1, \ldots \gamma_l\}$ under the new scheme.

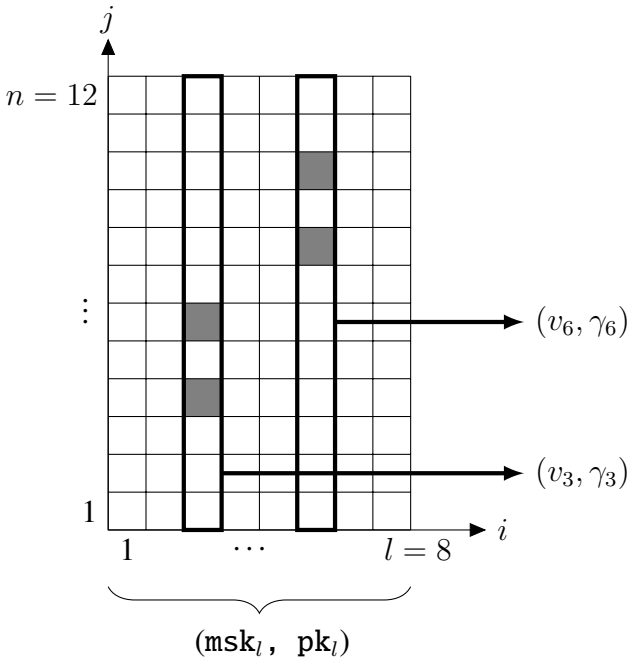


Figure 2.9: KAC key extension

As shown in Fig. 2.9, points lying on the same "column" (i value) are encrypted with the same keys and can be aggregated. Column 3 is associated with (v_3, γ_3) while column 6 is associated with (v_6, γ_6). Data points in different columns are encrypted with different keys and cannot be aggregated. Overall, the number of keys that Alice has to send to Bob is dependent on the number of "columns" which data points span.

The points $(3, 4), (3, 6), (6, 8), (6, 10)$ spans 2 columns, column 3 and 6. Hence 2 keys are released, $K_S = \{ \prod_{(3,j), j\in\{4,6\}} g_{n+1-j}^{\gamma_3}, \prod_{(6,j), j\in(8,10)} g_{n+1-j}^{\gamma_6} \}$. Notice that a single aggregate key $\prod_{(3,j), j\in\{4,6\}} g_{n+1-j}^{\gamma_3}$ is enough to decrypt both (3,4) and (3,6) because those two points have the

same i value, lying in the same column. Likewise, a single aggregate key $\prod_{(6,j),j\in(8,10)} g_{n+1-j}^{\gamma_6}$ is sufficient to decrypt $(6,8),(6,10)$.

2.4.3 Evaluation

KAC presents a viable scheme that can potentially reduce decryption key size since the aggregate key is constant in size. However, the linear-size `param` is a large storage overhead that has to be taken into consideration. The decryption process may also not be sufficiently efficient for large quantities of plaintext.

Chapter 3

Security Model

In this chapter, we will discuss our model proposed for secure storage outsourcing. We will also consider the runtime and security requirements of the system.

3.1 Entities

We first consider the entities involved in our proposed model. There are 4 entities involved.

Public Cloud (RC) A outsourced service provider hosting the encrypted files. It is assumed that RC has ample storage and bandwidth. In our model, RC merely provides storage services and has low computational capabilities.

Key Granter (KG) A trusted server responsible for key management, key distribution, user authentication and access mediation. In actual implementations, this is likely to be an internal server hosted by the organization. In our model, user authentication and access control is assumed to be handled by an external verifier which is not in the scope of our project.

Data Owner (DO) The author of the file containing time-series data. DO is responsible for key generation and file encryption. The keys generated will be uploaded to KG while the encrypted file will be uploaded to RC.

Authorized User (AU) A user of the system who is authorized to decrypt an encrypted file hosted in RC. Depending on access rights, AU may only be able to decrypt specific regions of the file.

3.2 Interactions

The entities operate with one another to perform tasks. These tasks can be classified into four main phases. The phases are **Synthesis**, **Upload**, **Retrieval** and **Key Distribution**. Fig. 3.1 illustrates the order of interactions between the various entities. We classify the interactions into 4 different phases.

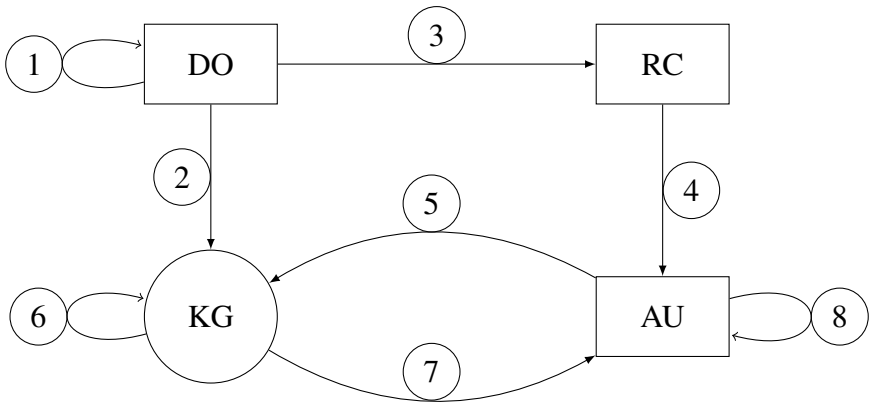


Figure 3.1: Interactions between entities

Synthesis

1. DO generates a set of master keys. Using the master keys, DO derives a set of symmetric frame keys and uses the frame keys to encrypt the video frames.

Upload

2. DO forwards the master keys as well as other essential parameters to KG.
3. DO uploads the encrypted video along with other public parameters/data to RC.

Retrieval

4. AU downloads the encrypted video and public data from RC.
5. AU requests for a meta-key from KG to decrypt specific video frames in set S.

Key Distribution

6. KG uses an external verifier to verify the identity as well as the access rights of AU. If AU indeed has rights to S, KG computes the meta-key for frames in S using the master key.
7. KG forwards meta-key to AU.
8. Using the meta-key from KG as well as public data from DO, AU is able to derive keys for video frames in S.

This model can be realized using encryption schemes such as KP-ABE, MRQED or KAC. DO uses these schemes to encrypt the frame keys, which are symmetric keys such as DES or AES. Next, the encrypted frame keys are uploaded along with the encrypted video. AU downloads the encrypted video frames along with the encrypted frame keys. The meta-key serves as a decryption key to the encrypted frame keys. Using the meta-key, AU can decrypt the encrypted frame keys, allowing for video decryption.

3.3 Performance Requirements

KG Bandwidth Since KG is hosted internally within the organization, bandwidth is at a premium. Therefore, the information transferred to/from KG should be kept to the minimal. As key transfers between KG to AU during **Key Distribution** are likely to occur frequently, the meta-key size should be of size $O(1)$. In **Upload** phase, keys are transferred from DO to KG and the bandwidth costs incurred should ideally be $O(\log(n))$.

KG Storage Maintaining a huge storage archive within the organization is costly. A huge data store also means that data is likely to be distributed over a few servers, increasing the chances of possible key leakage. Hence, the amount of key information stored for a single video is capped at $O(\log(n))$.

RC Bandwidth/Storage In **Upload**, DO sends the encrypted video frames and other relevant public data to RC. AU then downloads these data in **Retrieval** phase. Since RC is an outsourced storage provider, it is assumed that bandwidth and storage costs are cheap. Even so, the storage and bandwidth should be of the same order of magnitude as the size of the video (number of frames). Hence, both storage and bandwidth costs are capped at $O(n)$.

Encryption/Decryption Time Video encryption and decryption should be faster than the video's frame rate to meet real-time requirements. For purposes of this project, the frame rate of all video data is set to be 24 FPS. This also means that asymptotic runtimes of encryption and decryption are capped at $O(n)$.

3.4 Security Requirements

In this scheme, the only trusted entity is KG. The system should be secure against the following attacks.

Unauthorized data access The goal of the adversarial AU is to obtain frame keys that is not within his/her access rights. We do not consider attacks which are targeted at the external verifier (e.g. impersonating as another AU with more access rights).

Malicious/Spoofed RC The adversary is either a malicious RC or an entity spoofing as a trusted RC. He is able to communicate with the DO and AU and convince them that he is a trusted RC. His goal is to be able to extract plaintext information apart from metadata which is derivable from ciphertext (size of video, frame count).

KG Spoofing The adversary is able to communicate with DO and AU. His goal is to convince DO and AU that he is a trusted KG. By doing so, he is able to issue his own encryption/decryption keys, allowing him to gain access to the plaintext.

Collusion-resistant Bob and Charlie are AU with access rights to the set of keys S_B and S_C respectively. Bob and Charlie should not be able obtain key k where $k \notin S_B \cup S_C$.

Chapter 4

Preliminary Analysis

For this project, we have looked into the use of KAC (Chu et al., 2013) and KP-ABE (Lewko et al., 2010) as possible basis of implementations for our scheme. In this chapter, we present some in-depth performance analysis and comparisons between these two schemes.

All empirical data are gathered from prototypes built using Charm (Akinyele, Garman, Miers, Pagano, Rushanan, Green, & Rubin, 2013) cryptographic framework, running on Macbook Air 2013 (Intel Core-i7-4650u). Symmetric pairings over Type-A (supersingular) curves with a base field of 512 bits are chosen as the underlying pairing group. No pre-computation tables were used for group exponentiation.

4.1 Space Requirements

4.1.1 Meta-Key

The meta-key or aggregate key is an important part of our system. The meta-key allows the user to decrypt the symmetric frame keys stored in RC, which can then be used to decrypt the video frames.

In KP-ABE, using the "bag of bits" notation introduced in Section 2.2.7, it is possible to construct a meta-key that is logarithmic in size for a contiguous range[1] of frames. Hence, MK generated by the KP-ABE's **KeyGen** algorithm can be constructed with just $O(log(n))$ group elements. In KAC, the aggregate key (K_S) generated by the **Extract** algorithm serves as the meta-key. K_S is constant in size, comprising of a single group element.

Fig. 4.1 compares differences in key size with varying video size for decryption of a single contiguous range. For KP-ABE, the key size displays a logarithmic growth rate with respect to n while KAC generates constant-size keys for all values of n.

[1]Sequence in the form of $\{i, i+1, i+2, \dots\}$

For meta-keys that are non-contiguous, KP-ABE may produce keys of $O(|S|)$ in size because of the need to encode individual values as separate attributes, with S representing the set of frames that the meta-key is authorized to decrypt. The "bag of bits" notation can only be used for represent contiguous ranges. KAC on the other hand is able to consistently produce keys of size $O(1)$ even for non-contiguous values. For simplicity of analysis, we shall be comparing storage and runtime performance that is of contiguous in nature in the subsequent sections.

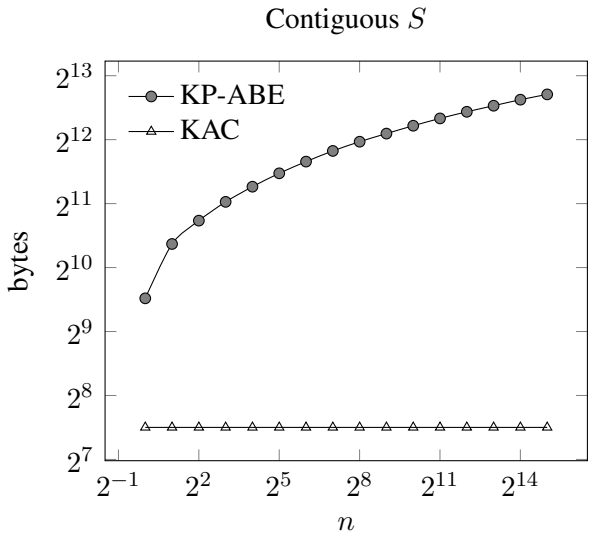


Figure 4.1: Meta-key size

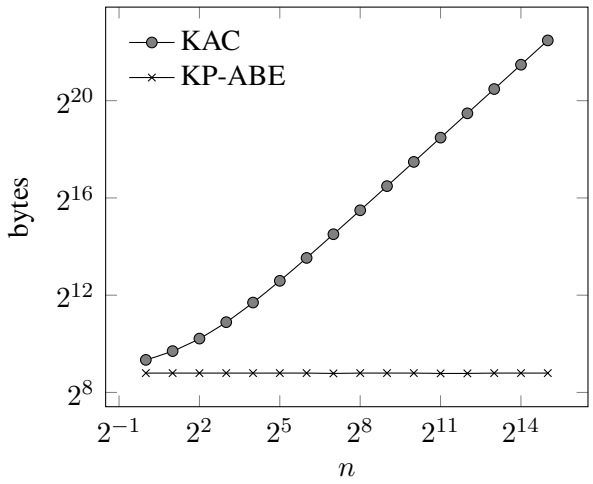


Figure 4.2: KG Storage

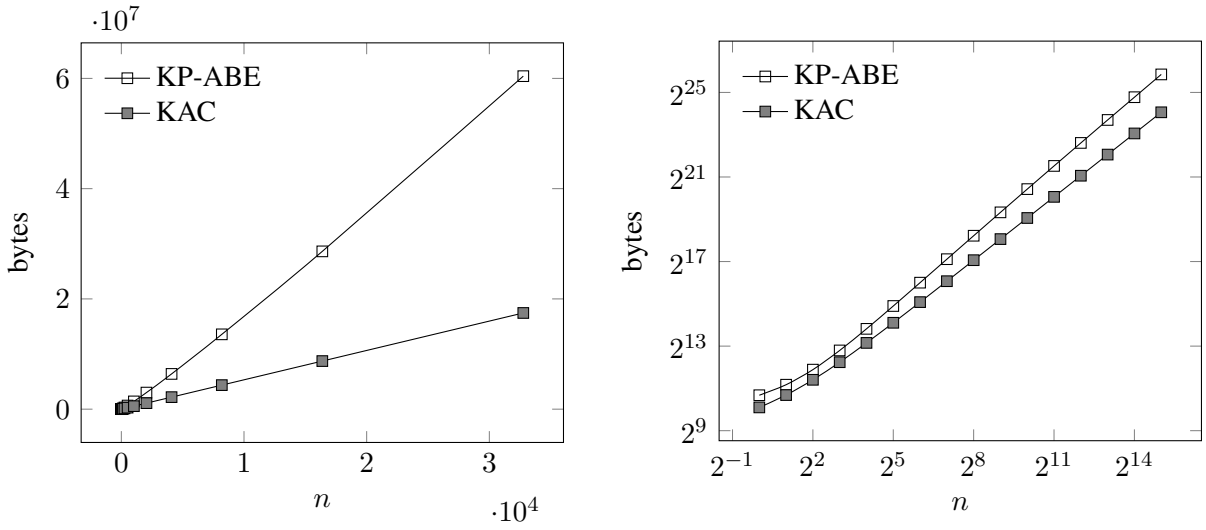
4.1.2 KG – Private Storage

The KG has to contain a minimal amount of parameters for key computation and these parameters have to be stored in KG.

	Master Key	Public Key	Public Parameters
KAC	1 prime element	1 group element	$2n$ group elements
KP-ABE	3 prime elements	5 group elements	–

Table 4.1: KG storage requirements

KP-ABE's **Key Generation** algorithm requires the only master key and public key as inputs, and these 2 inputs are constant in size. KAC's **Extract** algorithm, which is analogous to KP-ABE's **Key Generation**, requires the master key and a linear-size public parameter `param`. As summarized in Table 4.1, because of the need to store a linear-size parameter required for the generation of the meta-key, the amount of storage required by KAC is much more than



(a) Linear Scale

(b) Log-Log scale

Figure 4.3: RC storage

KP-ABE. In Fig. 4.2, we see that KG storage required by KG increases to more than 2^{20} for $n > 2^{12}$. In this aspect, KP-ABE is a better choice as opposed to KAC.

4.1.3 RC – Public Storage

	Public Key	Ciphertext (per Frame)	Public Parameters
KAC	1 group element	3 group elements	$2n$ group elements
KP-ABE	5 group elements	$O(\log(n))$ attributes and group elements	–

Table 4.2: RC storage requirements

RC is where we store the encrypted video, as well as the encrypted symmetric keys for the video frames. As summarized in Table 4.2, the use "bag of bits" notation results in logarithmic-size ciphertext. Hence, KP-ABE produces ciphertext that is $O(\log(n))$ in $O(n \log(n))$ storage for n frames. Ciphertext generated by KAC is constant size, resulting in $O(n)$ storage for n frames. Fig. 4.3 shows the deviation in storage for KP-ABE and KAC for increasing n. The difference in storage is obvious when plotted against a linear scale, as shown in Fig. 4.3a.

4.2 Runtime

4.2.1 Setup

	Pairing	Exponentiation	Multiplication	Division	Hash
KAC	1	$2(2n-1)+1=\Theta(n)$	0	0	0
KP-ABE	$O(1)$	$O(1)$	$O(1)$	0	0

Table 4.3: Running complexity for the setting up of parameters

Both KAC and KP-ABE schemes have a setup phase to generate the parameters required for encryption/decryption. As shown in Table 4.3, the **Setup** phase of KP-ABE takes only constant time, generating private and public keys of size $O(1)$. The **Setup** phase in this case will be performed by DO. DO encrypts the symmetric keys of the video using KP-ABE. Since only the public key is transferred from DO to KG, the bandwidth required is $O(1)$.

For KAC, the setup process consists of **Setup** and **KeyGen** algorithms. The **Setup** phase is where the computation of `param` takes place while KAC's **KeyGen** algorithm is analogous to the **Setup** algorithm of KP-ABE where the generation of public and private keys take place. Since DO is required to transfer `param` to KG, $\Theta(n)$ bandwidth is required. Also, due to the need to generate $2n$ group elements, KAC incurs a larger runtime complexity than KP-ABE.

Instead of transferring $\Theta(n)$ size `param` to KG, DO could transfer a constant-size α to KG, where α is a secret exponent used in KAC's **Setup** algorithm to generate `param`. KG then generates $2n$ group elements from α. However, this method limits the number of DOs KG can support since generating `param` is computationally expensive.

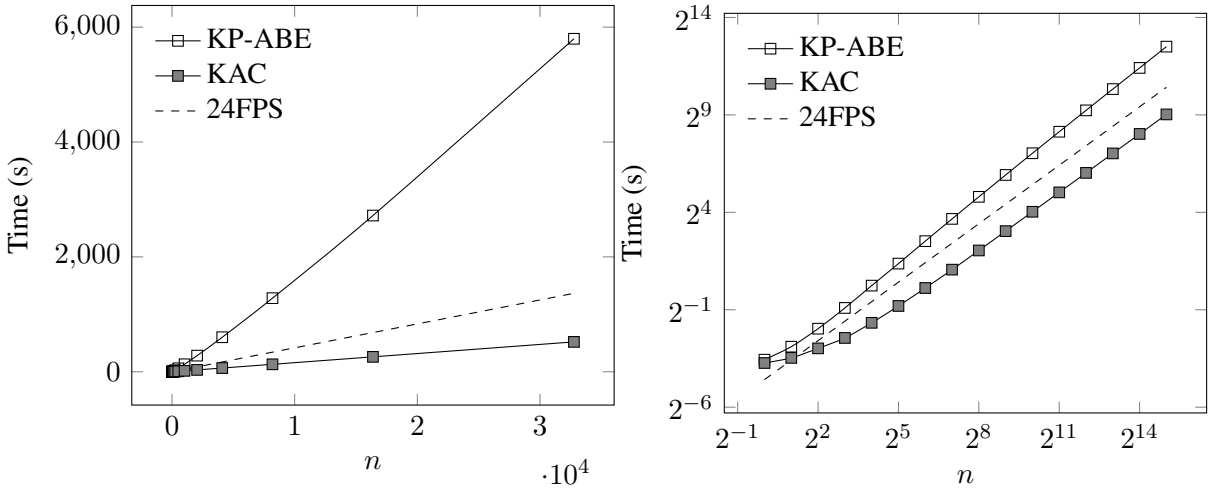
4.2.2 Encryption

	Pairing	Exponentiation	Multiplication	Division	Hash
KAC	1	3	2	0	0
KP-ABE	0	$O(log(n))$	$O(1)$	0	$O(log(n))$

Table 4.4: Encryption complexity per frame

Table 4.4 summarizes the number of operations it takes for both setup and encryption. We see that KP-ABE has a runtime complexity of $O(\log(n))$ to encrypt a single piece of ciphertext. This translates to $O(n\log(n))$ for n frames. KAC on the other hand, takes constant time to encrypt each piece of plaintext, resulting in a total complexity of $\Theta(n)$ for n frames. In Fig. 4.4a

and Fig. 4.4b which shows the time it takes for both setup and encryption, we see that there is a divergence in the runtime for large n values, especially in Fig. 4.4a where a linear scale is used. In fact, we see that KAC is efficient enough to be able to encrypt keys for a 24FPS video in real-time.



(a) Linear Scale

(b) Log-Log scale

Figure 4.4: Setup + Encryption time

4.2.3 Meta-key Generation

	Pairing	Exponentiation	Multiplication	Division	Hash
KAC	0	1	$\|S\|$	0	0
KP-ABE	0	$O(\log(n))$	$O(\log(n))$	0	$O(\log(n))$

Table 4.5: Meta-key generation complexity (Contiguous Range)

The response time to AU's request depends on the time it takes for KG to generate the meta keys. In Table 4.5, we see that KAC's **Extract** algorithm is dependent on S unlike KP-ABE's **KeyGen** algorithm. Since $|S| \leq n$, the runtime complexity of KAC may be as high as $O(n)$. In Fig. 4.5, we see that the actual runtime of KP-ABE is much slower than than KAC, despite KAC having a larger complexity. This is because KP-ABE requires $O(\log(n))$ exponentiations which are more computationally intensive than multiplications.

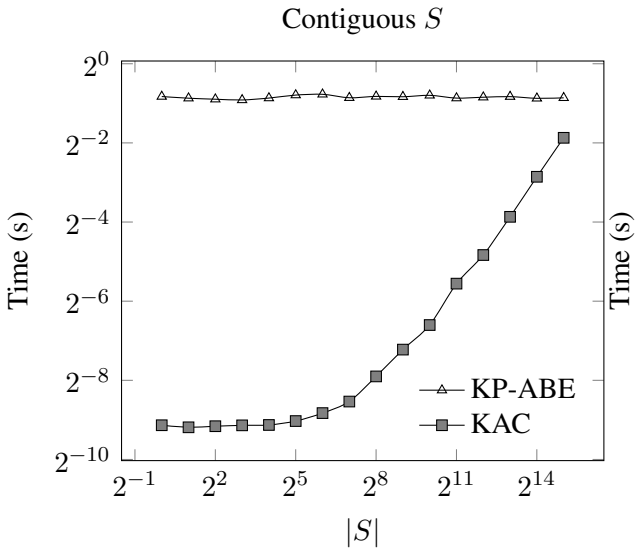


Figure 4.5: Meta-key generation time (contiguous range)

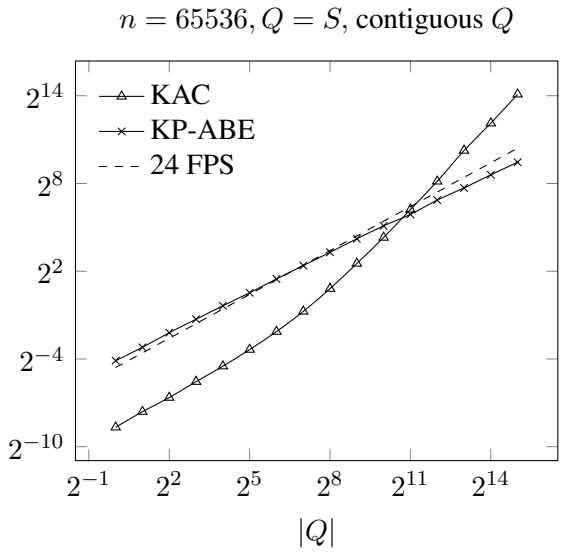


Figure 4.6: Decryption Time

4.2.4 Decryption

	Pairing	Exponentiation	Multiplication	Division	Hash
KAC	2	0	$2\|S\| + 1$	1	0
KP-ABE	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

Table 4.6: Decryption complexity per frame

AU may request for a meta-key that is able to decrypt keys in the set S. Instead of decrypting all of the frame keys in S, the AU may choose to decrypt just a subset of the keys. We denote this decryption set as Q, where $Q \subseteq S$. In Table 4.6, we see that decryption for KAC has a complexity of $\Theta(|S|)$ per ciphertext, or $|Q|\Theta|S| = O(|S^2|)$ for decryption of $|Q|$ frames. In comparison, KP-ABE requires $O(\log(n))$ per ciphertext, or $|Q|O(\log(n))$ for $|Q|$ frames. Experimental results in Fig. 4.6 show that the runtime of KAC begins to exceeds KP-ABE for $|Q| > 2^{12}$ when KAC's multiplications outpaces KP-ABE's operations. For $|Q| > 2^8$, we see that KP-ABE's decryption rate outpaces the frame rate of a 24FPS video.

4.3 Conclusion

Both KAC and KP-ABE have their strengths and weaknesses in both storage and runtime (Table 4.7). KAC has the advantage of producing meta-keys of size $O(1)$. KP-ABE on the other hand has a small storage footprint on KG, requiring only $O(1)$ storage. However, its construction based on access trees also means that it has disadvantages similar to GGM-tree. KP-ABE

	KG	RC	Meta-key (KG $\rightarrow$ AU)
KAC	$\Theta(n)$	$\Theta(n)$	$O(1)$
KP-ABE	$O(1)$	$O(n\log(n))$	Contiguous: $O(\log(n))$ Non-contiguous: $O(\|S\|)$

(a) Storage

	Setup + Encryption	Decryption	Meta-key generation
KAC	$\Theta(n) + \Theta(n) = \Theta(n)$	$\|Q\|\Theta(\|S\|)$	$\Theta(\|S\|)$
KP-ABE	$1 + O(n\log(n)) = O(n\log(n))$	$\|Q\|O(\log(n))$	$O(\log(n))$

(b) Runtime

Table 4.7: Summary of storage and runtime requirements

is also able to provide support for multi-dimensional data since key produced is collusion-resistant. Despite the slower decryption speed and large KG storage, we have chosen KAC as the implementation basis for our scheme. KAC is chosen due to the advantage of having a constant-size meta-key and small ciphertext size as well as for its algorithmic simplicity which allows for greater optimization flexibility. We will be discussing strategies to overcome some of the KAC's weaknesses in Chapter 5.

Chapter 5

Proposed System

In this chapter, we will be looking into various optimization strategies which help mitigate the shortcomings of KAC. In Section 5.1, we describe a KAC-GGM hybrid scheme which is able to attain faster contiguous decryption by decreasing data granularity. Section 5.2 discusses optimizations in the KAC's **Decrypt** algorithm that allows for fast decryption for arithmetic sequences. In Section 5.3, we consider an verification protocol based on homomorphic MAC which reduces bandwidth costs for transfers from DO to KG as well as KG's storage footprint from $\Theta(n)$ to $O(1)$. Finally in Section 5.4, we propose a construction that provides support for fast decryption of layered content, for instance, different resolutions for the same video clip. An analysis of the various proposed algorithms and a summary of how these algorithms are used in our model will be presented in Chapter 6.

5.1 KAC-GGM Hybrid

In this section, we discuss KAC-GGM hybrid schemes which improves decryption runtime. Section 5.1.1 discusses the use of GGM to generate fixed video chunks of size m to reduce the amount of KAC decryptions needed to be performed. In Section 5.1.2, we propose a generalized construction of the scheme that reduces decryption runtime significantly.

5.1.1 Fixed Chunking

A simple solution to improve decryption runtime is to decrease data granularity. Instead of encrypting each frame under its own distinct ciphertext class, we encrypt the video in short chunks, each chunk containing m frames. With chunking, the amount of plaintext and ciphertext is reduced, improving storage, encryption and decryption overheads. However, there are security concerns when granularity is decreased. We look at an example to illustrate the problem.

Suppose that $n = 1024, m = 8$ and AU is only entitled to frames in $[6, 507]$. Since $m = 8$, we will obtain the following set of video chunks, $\{[1, 8], [9, 16], \ldots, [505, 512], \ldots, [1017, 1024]\}$. By decrypting chunk $[1, 8]$, the AU is able to obtain $[6, 8]$ which he/she has legitimate access to. However, $[1, 5]$ will be also be revealed to AU since $[1, 5] \subset [1, 8]$. Similarly, decrypting chunk $[505, 512]$ will reveal $[508, 512]$.

Using GGM trees to aggregate frames in each video chunk, we are able to maintain the confidentiality of frames outside of AU's authorized set. Instead of encrypting the entire chunk of m frames using KAC, KG first generates n/m GGM root nodes. KG then encrypts these root nodes under its own ciphertext class/index using KAC. Each GGM root node is used to derive m pseudorandom sequences which serves as the frame keys. The construction is illustrated in Fig. 5.1.

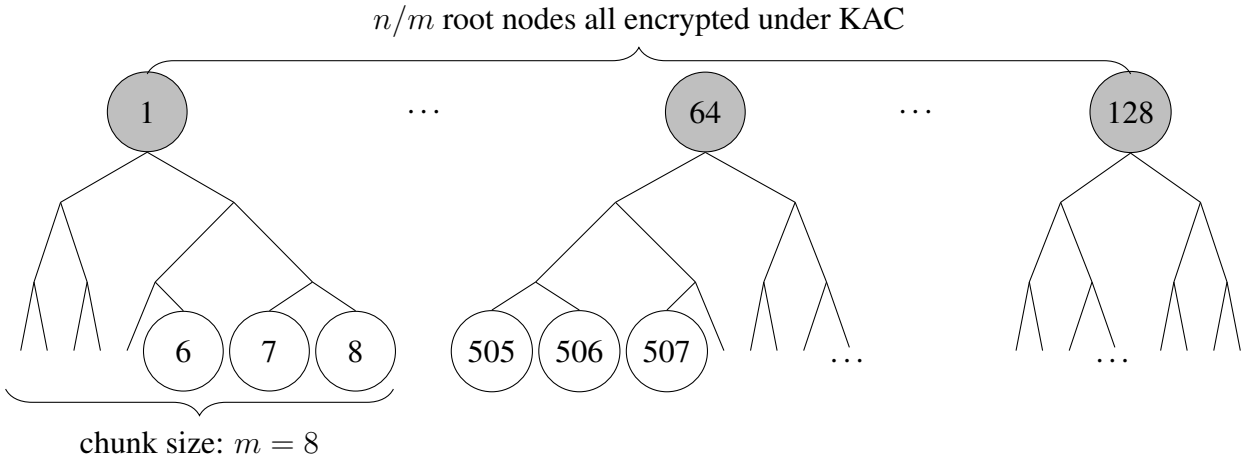


Figure 5.1: KAC-GGM hybrid structure

To release keys for $[6, 507]$, KG sends AU an KAC aggregate key $K_S, S = \{2, 3, \ldots, 62, 63\}$. This allows AU to decrypt GGM roots and derive the keys for $[9, 504]$. GGM roots $1, 64$ are not included in S as it will allow AU to derive keys outside of $[6, 507]$. KAG builds the GGM tree and releases the inner nodes needed to derive the keys for $[6, 8]$ and $[505, 507]$, in a manner similar to GGM trees illustrated in Fig. 2.2.

5.1.2 Variable Chunking

An alternate method of constructing the scheme explained in Section 5.1.1 is to use a single GGM tree. As seen in Fig. 5.2, KG first uses a GGM root node to derive n leaf nodes. The monolithic tree is decomposed into n/m subtrees, each subtree having m leaf nodes. KG then encrypts the root node of each subtree under KAC, represented by the shaded nodes.

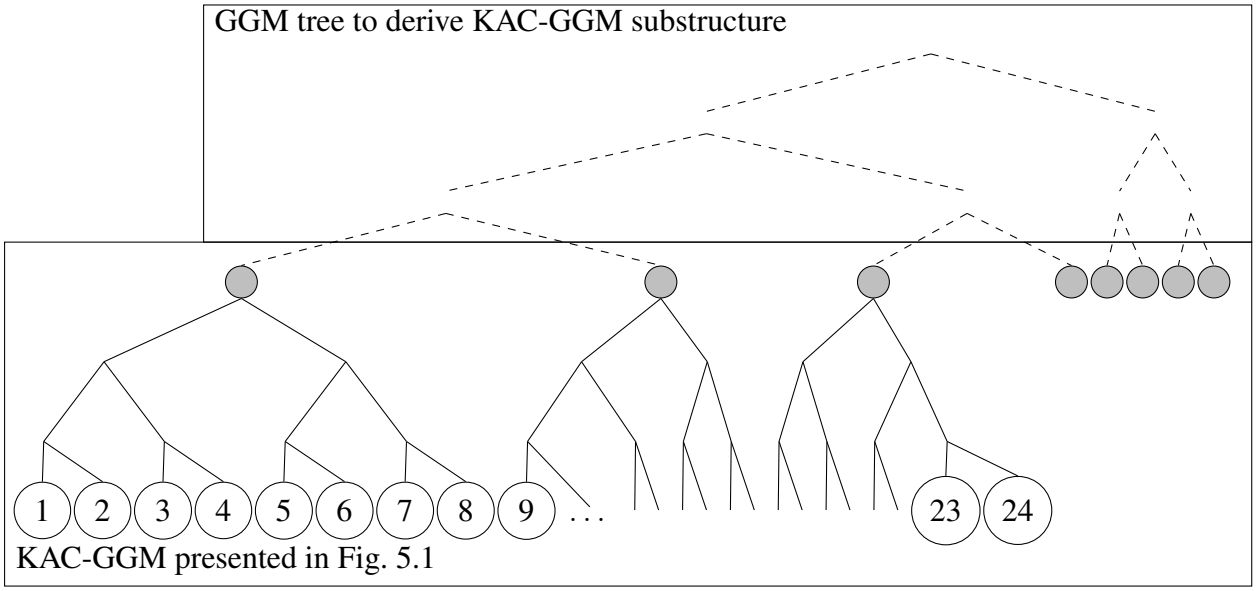


Figure 5.2: Generalization of construction shown in Fig. 5.1

Instead of using fixed size chunks, we are able to generalize the KAC-GGM construction to support variable chunking. We illustrate variable chunking with some examples. To decrypt $[1, 1024]$, the ideal chunk size is $m = n = 1024$. Only a single GGM node is required to generate all frame keys. To decrypt $[1, 522]$, we can divide the range into subranges $[1, 512]$, $[513, 520]$ and $[521, 522]$, yielding ideal chunk sizes $m_1 = 512, m_2 = 8, m_3 = 2$ respectively. Using such decomposition methods, AU only requires 3 GGM nodes to derive $[1, 512]$.

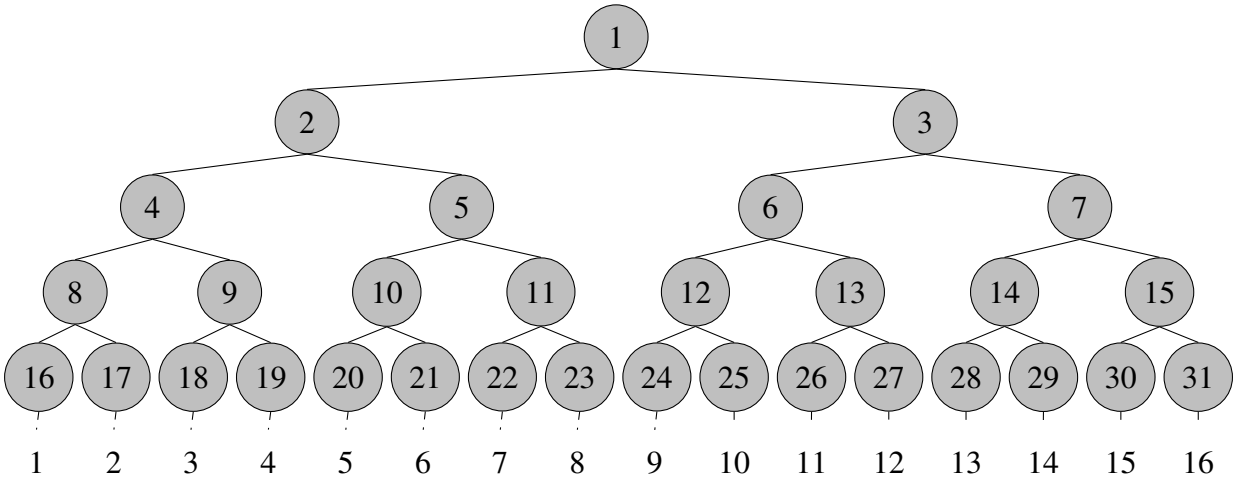


Figure 5.3: Generalized KAC-GGM (All GGM nodes are encrypted under KAC)

The construction for variable chunking is the same as described in Fig. 5.2 with a slight difference. Instead of encrypting only a single tree stratum, we now encrypt every node in the GGM tree, as shown in Fig. 5.3. Each shaded node is encrypted using its own unique KAC ciphertext class numbered in a breadth-first fashion. To release keys for for $[3, 8]$, KG

simply forwards $K_{\hat{S}}$, where $\hat{S} = \{9, 5\}$ to AU. AU uses $K_{\hat{S}}$ to decrypt nodes $9, 5$. AU then decrypts node 9 and node 5 using KAC. After decryption of the GGM nodes, AU is able to reconstruct the GGM subtrees with node 9 and 5 as the root, deriving frame keys for $[3, 4]$ and $[5, 8]$ respectively. As compared to fixed chunking, variable chunking results in even lower granularity, which translates to higher decryption speeds.

5.2 KAC Decryption Optimizations

The construction of KAC's **Decrypt** algorithm provides a lot of opportunities for optimization. The decryption of a single ciphertext ($\mathfrak{C}_i$) under KAC is defined as:

$$m_i = c_{i_3} \cdot \hat{e}(K_S \cdot \prod_{j \in S, j \neq i} g_{n+1-j+i},\, c_{i_1}) \,/\, \hat{e}(\prod_{j \in S} g_{n+1-j}, c_{i_2})$$

where $\mathfrak{C}_i = \langle c_{i_1}, c_{i_2}, c_{i_3} \rangle$. We denote $\rho_i = \prod_{j \in S, j \neq i} g_{n+1-j+i}$ and $\lambda = \prod_{j \in S} g_{n+1-j}$.

5.2.1 Decryption under same K_S

For decryptions under the same K_S, the value of λ remains unchanged as it is independent of i. λ can be computed once and reused for subsequent decryptions instead of recomputing it for Q times. This simple strategy can save a total of $|S|(|Q| - 1)$ multiplications.

5.2.2 Optimization for varying frame rates

In the KAC-GGM hybrid scheme presented in Section 5.1, higher decryption speed is achieved by creating chunks of contiguous frames. However, the chunking strategy is not useful if we want to decrypt non-contiguous frames, especially frames which are part of an arithmetic sequence. Optimizing decryption for arithmetic sequences is especially useful if we want to support video decryption of varying frame rates. For instance, AU with limited access rights may only be allowed to decrypt the video only at $\frac{1}{8}$ of the original frame rate. This translates to the set of frames belonging to the arithmetic sequence, $a_i = a_1 + 8(i - 1), i \geq 1$. We are able to speed up the computation of ρ for arithmetic sequences and support fast decryption of varying frame rates.

Suppose S and Q are single arithmetic sequences where, $S = \{x, x + d, x + 2d, \ldots, x + d(|S| - 1)\}, Q = \{i, i + d, i + 2d, \ldots, i + d(|Q| - 1)\}, Q \subseteq S$.

For the the message encrypted under i^{th} ciphertext class (m_i), we have

$$\begin{aligned}\rho_i &= \prod_{j \in S, j \neq i} g_{n+1-j+i} \\ &= \Big(\prod_{j=0}^{|S|-1} g_{n+1-(x+jd)+i}\Big) / g_{n+1}\end{aligned}$$

To compute m_{i+d}, AU is able to make use of ρ_i.

$$\begin{aligned}\rho_{i+d} &= \prod_{j \in S, j \neq i+d} g_{n+1-j+(i+d)} \\ &= \Big(\prod_{j=0}^{|S|-1} g_{n+1-(x+jd)+(i+d)}\Big) / g_{n+1} \\ &= \frac{g_{n+1-x+i+d}}{g_{n+1-(x+d(|S|-1))+i} \times g_{n+1}} \times \prod_{j=0}^{|S|-1} g_{n+1-(x+jd)+i} \\ &= \frac{g_{n+1-x+i+d}}{g_{n+1-(x+d(|S|-1))+i}} \times \rho_i\end{aligned}$$

AU then applies this optimization recursively for $\rho_{i+2d}, \rho_{i+3d}, \ldots, \rho_{i+d(|Q|-1)}$ instead of re-computing ρ each time. Substantial speedups have been observed from actual experiments. These results will be presented in Section 6.2.

5.3 Homomorphic MAC

One of the drawbacks of using KAC is the $\Theta(n)$ size `param` that is required to be stored in KG. In this section, we introduce a protocol that alleviates this issue, reducing the amount of private storage needed from $\Theta(n)$ to $O(1)$.

The aggregate key is computed as $K_S = \prod_{j \in S} g^{\gamma}_{n+1-j}$, where γ is the `msk`. Hence, we are able to delegate the computation of $\prod_{j \in S} g_{n+1-j}$ to the AU instead, especially since AU would have already obtained $\texttt{param} = \langle g, g_1, g_2, \ldots, g_n, g_{n+2}, \ldots, g_{2n} \rangle$ from RC which is required for running KAC's **Decrypt**.

1. AU has rights to access frame keys of set S.
2. AU obtains `param` from RC.
3. AU computes and submits $\langle \prod_{j \in S} g_{n+1-j}, S \rangle$ to KG.

4. KG verifies that AU is entitled to frames of set S.
5. KG computes $K_S = \prod_{j \in S} g_{n+1-j}^{\gamma} = (\prod_{j \in S} g_{n+1-j})^{\gamma}$.
6. KG transfers K_S to AU.
7. AU uses K_S and `param` to decrypt the frames in S.

With such a verification protocol, there is no need for KG to store the linear-size `param` as part of the computation of K_S is delegated to AU. KG is only responsible for exponentiating the product of the group elements in `param` with `msk`. However, this scheme is vulnerable to spoofing attacks, as there is no way for the server to verify AU's computation.

Consider Harry, a malicious AU, who only has authorized access to keys in set S. He is however interested in obtaining keys in set U, $U \not\subseteq S$.

1. Harry computes and sends KG $\langle \prod_{j \in U} g_{n+1-j}, S \rangle$.
2. KG verifies that Harry have access to S.
3. KG computes and transfers $K_U = (\prod_{j \in U} g_{n+1-j})^{\gamma}$ to Harry.
4. Harry has managed to convince the server to K_U. Using K_U, he is able to generate frame keys in U which he does not have the authority to access.

To prevent such attacks, KG needs to be able to verify AU's computation to ensure that it is indeed $\prod_{j \in S} g_{n+1-j}$ and not a spoofed value.

5.3.1 Verification Protocol

In order to verify AU's computation, we introduce a protocol based on homomorphic MAC construction which allows KG to verify the AU's computation without the need to store `param`. The protocol consists of 3 algorithms, **Signing**, **Computation**, **Verification**.

Signing DO signs the group elements in `param` which used to generate the aggregate key, K_S. We denote this set of signatures as `param_sig`.

1. DO select random values $\beta, \beta_1, \beta_2, \ldots, \beta_n \in \mathbb{Z}_p$.
2. DO extracts $n+1$ group elements $g, g_1, \ldots, g_n$ from `param`. g is the generator for $\mathbb{G}$, while $g_1, g_2, \ldots, g_n$ are the group elements used in the computation of K_S.
3. DO computes `param_sig` $= \langle g_1^{\beta} g^{\beta_1}, g_2^{\beta} g^{\beta_2}, \ldots, g_n^{\beta} g^{\beta_n} \rangle = \langle a_1, a_2, \ldots, a_n \rangle$, the homomorphic signature for $\langle g_1, \ldots, g_n \rangle$.

Computation AU computes the necessary parameters required for the generation for an aggregate key and forwards it to KG.

1. AU has access rights to frame keys of set S.
2. AU obtains `param` and `param_sig` from RC.
3. AU computes and submits $\langle w_1, w_2, w_3 \rangle = \langle \prod_{i \in S} g_{n+1-i}\ , \prod_{i \in S} a_{n+1-i}, S \rangle$ to KG.

Verification KG verifies $\langle w_1, w_2, w_3 \rangle$. K_S is issued only if w_1, w_2 are computed using the elements in `param` and `param_sig` corresponding to set w_3.

1. KG computes $\texttt{aggregate\_sig} = w_1^{\beta} \times \prod_{i \in w_3} g^{\beta_{n+1-i}}$.
2. KG computes the $K_{w_3} = w_1^{\gamma}$ only if $\texttt{aggregate\_sig} = w_2$.

This scheme still requires DO to send $\beta, \beta_1, \beta_2, \ldots, \beta_n$ to KG, consuming $\Theta(n)$ bandwidth. Instead of storing them permanently, DO could choose to generate $\beta_1, \beta_2 \ldots \beta_n$ using a random oracle from β. Therefore, DO only has to send constant-size β to KG. KG can then derive $\beta_1, \beta_2 \ldots \beta_n$ from β. A security proof of the protocol will be presented in Section 6.3.1.

5.4 Provisioning for Layered Content

It has become increasingly common for video files to be encoded with different resolutions. This strategy is commonly employed in video streaming sites to cater to clients with varying network bandwidth. In our case, different AUs are granted access rights to footage of different resolutions. In this section, we explore mechanisms that support such capabilities.

We consider a video file with two quality levels, high quality (HQ) and low quality (LQ). Borrowing the ideas from Section 2.2.9, we can segment the video into two layers, a LQ layer and a HQ enhancement layer. LQ video frames correspond to the LQ layer. To obtain HQ video frames, AU will need a combination of data from both LQ and HQ enhancement layer. In other words, the HQ enhancement layer contains information which is present in HQ frames but missing in LQ frames. This use of an enhancement layer reduces the amount of storage needed as there is no duplication of information between LQ and HQ frames. The HQ enhancement layer is higher in privilege compared to LQ layer, denoted as $HQ > LQ$.

Fig. 5.4 features a combination of KAC-GGM with hash chaining which is described in Section 2.2.9 and Equation 2.1. AU has access rights to frame keys $[(HQ, 1), (HQ, 8)]$. Since $HQ > LQ$, AU is able to retrieve keys in $[(LQ, 1), (LQ, 8)]$ by computing $(LQ, i) = h((HQ, i)||\text{padding})$, where h is a hash function and i represents the frame number, $1 \leq i \leq n$.

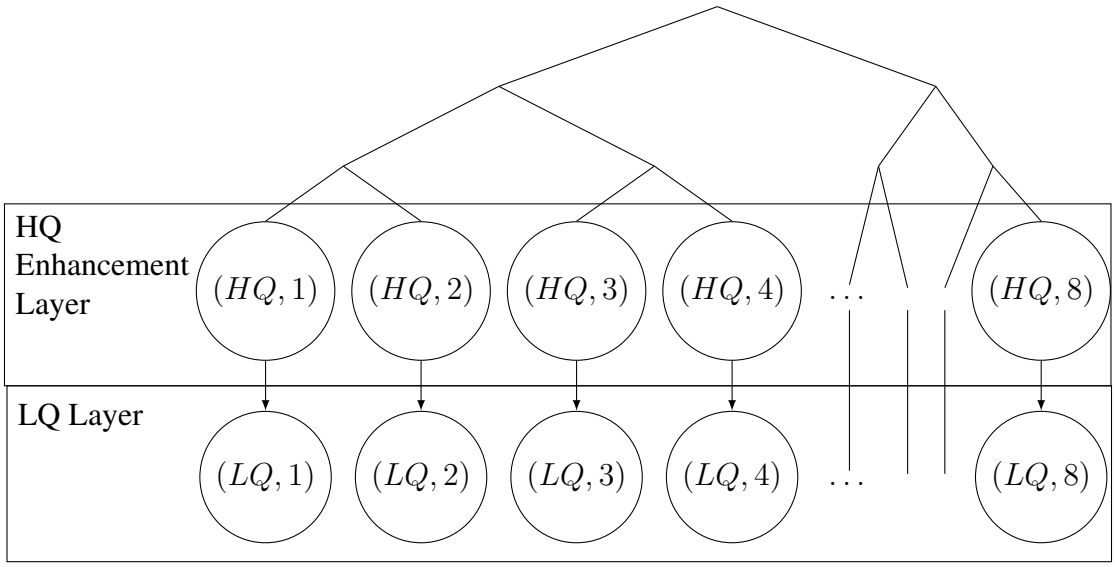


Figure 5.4: Layered hashing

By hashing the keys in the upper layer, AU is able to obtain the corresponding keys in the lower layers.

Now suppose that AU only has access to $[(LQ, 1), (LQ, 8)]$. It is clear from the Fig. 5.4 that we are unable to aggregate the LQ frame keys under the root as doing so exposes HQ frame keys. Consequently, decryption of the LQ layer can only be performed at the leaves of the tree which is highly inefficient. In fact, all layers except the highest privileged layer have to be decrypted at the leaves. Hence, hash chaining is not a viable construction for encryption of layered content.

5.4.1 Layered Symmetric Encryption

A simple approach to support layered decryption is to generate l KAC-GGM hybrid trees, 1 tree for each layer. To decrypt l layers, the AU runs KAC-GGM hybrid algorithm l times. However, running KAC's **Decrypt** algorithm l times can be slow.

Instead of using KAC to decrypt each layer, we can instead choose to use symmetric decryption to generate the frame keys for the lower layers, at a cost of additional public storage in RC. In addition to generating l KAC-GGM hybrid structures, DO also encrypts the frame keys of layer $(i-1)$ using the frame keys of layer i as the symmetric key, where i represents the privilege level.

For l layers, we have $E_{k_{(i+1,j)}}(k_{(i,j)}), 1 \leq i \leq l, 1 \leq j \leq n$, where j represents frame number and i represents the privilege level. This process gives us $l-1$ intermediate layers. Decrypting all l layers involves KAC decryption of the highest layer, followed by symmetric decryption of $l-1$ intermediate layers.

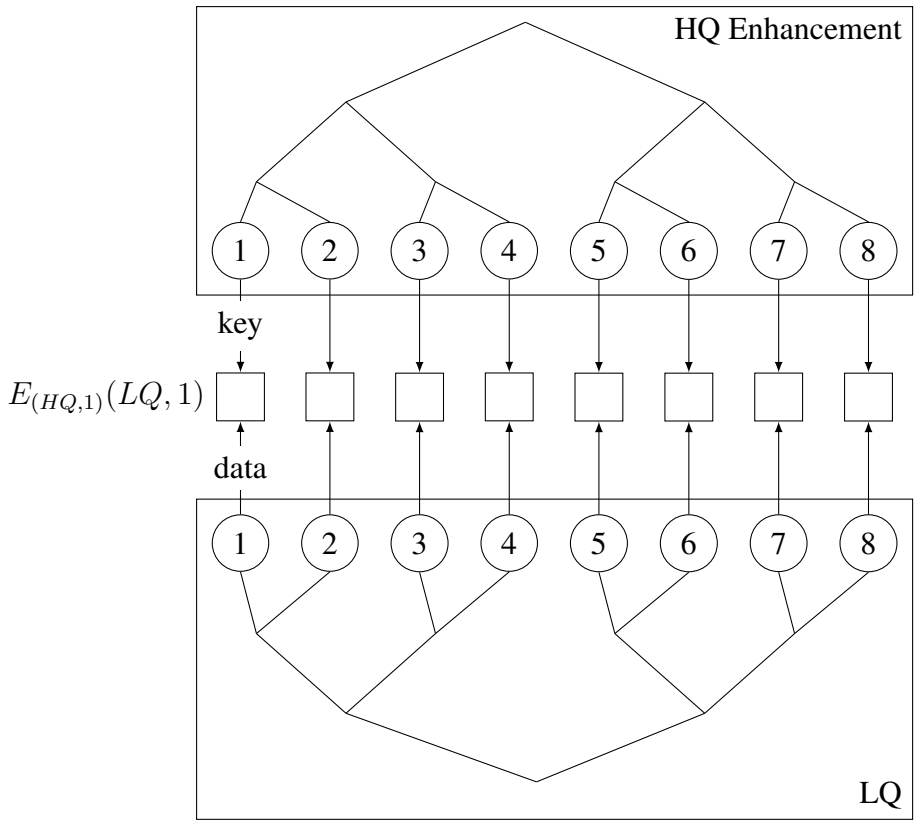


Figure 5.5: Layered encryption using intermediate layers

An example involving just 2 layers is shown in Fig. 5.5. LQ frame keys are encrypted by the frame keys of the HQ layer, forming an intermediate encrypted layer. For instance for frame 1, $(LQ, 1)$ is encrypted with using $(HQ, 1)$ as the symmetric key, forming $E_{(HQ,1)}(LQ, 1)$. The intermediate layer is stored in the RC along with the KAC. To obtain high resolution frames in range $[2, 4]$, AU first derives keys in $[(HQ, 2), (HQ, 4)]$ using KAC. AU then uses symmetric decryption to obtain $[(LQ, 2), (LQ, 4)]$.

Chapter 6

Analysis

In this chapter, we present storage, runtime, as well as security analysis of our proposed techniques in Sections 6.1 to 6.4. In Section 6.5, we will look at how our proposed techniques can be applied to our model to fulfill performance and security requirements. Similar to Chapter 4, experiments are performed on prototypes built using Charm (Akinyele et al., 2013) cryptographic framework, running on Macbook Air 2013 (Intel Core-i7-4650u). Symmetric pairings over Type-A (supersingular) curves with a base field of 512 bits are chosen as the underlying pairing group for KAC. For the implementation of GGM trees, SHA-256 was used as the underlying CSB, where $G_0(x) =$ SHA-256$(x \,||\, \text{`}l\text{'})$ and $G_1(x) =$ SHA-256$(x \,||\, \text{`}r\text{'})$.

6.1 KAC-GGM Hybrid

In this section, we evaluate the runtime and storage performance of both fixed and variable chunking in a KAC-GGM hybrid scheme.

6.1.1 Fixed Chunking

	KAC	Fixed Chunking	Entity
$\mathfrak{C}$	$\Theta(n)$ group elements	$\Theta(n/m)$ group elements	RC
`param`	$2n$ group elements	$2n/m$ group elements	RC, KG
GGM Root	-	n/m hash digests	KG
Meta-key	$O(1)$	Contiguous S: $O(1) + 2\,O(\log(m))$ Non-contiguous S: $O(\|S\|)$	KG $\rightarrow$ AU

Table 6.1: Storage comparison between KAC and KAC-GGM (Fixed Chunking)

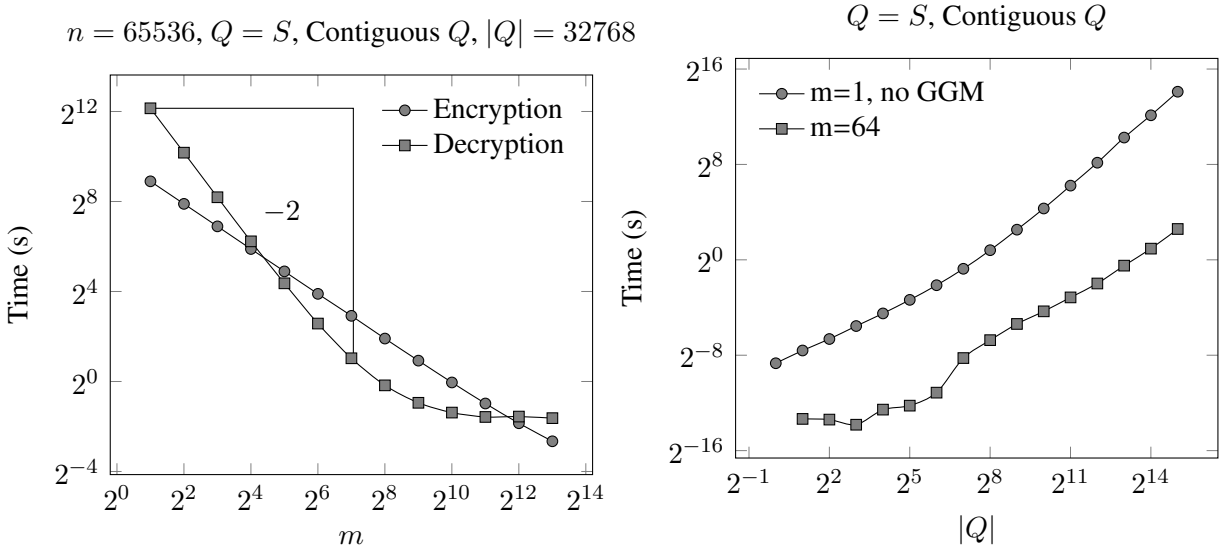
In Table 6.1, we see that for contiguous decryption, KAC-GGM incurs a meta-key size of $O(1) + 2\,O(\log(m))$. $O(1)$ being the size of the aggregate key K_S and $2\,O(\log(m))$ being the

extra key material derived from the GGM tree in the start and end of the range, as illustrated in Fig. 5.1.

For non-contiguous decryption, none of the keys can be aggregated. Therefore, KG has to send $O(n)$ leaf nodes to AU.

The total amount of storage required by RC has been reduced from $\Theta(n)$ to $\Theta(n/m)$ due to the reduction in ciphertext classes. The storage requirements of KG on the other hand, have changed from $2n$ group elements to $2n/m$ group elements and n/m hash digests. Since $m \geq 2$, there is a net reduction of storage in KG.

Empirical results in Fig. 6.1a show that running time of encryption exhibits linear speedup with respect to increasing m. Contiguous decryption complexity for fixed chunking is calculated to be $(|Q|/m)\,\Theta(|S|/m) = O(|S|^2/m^2)$. The plot has a gradient of -2 for $m \leq 2^7$. This translates to a speedup of m^2, consistent with our theoretical calculations. For $m > 2^7$, runtime gains are sub-optimal. This can be attributed to GGM tree generation dominating over the runtime of KAC decryption.



(a) KAC-GGM Encrypt/Decrypt Time

(b) KAC-GGM Decryption Speedup

Figure 6.1: Empirical results of KAC-GGM

Fig. 6.1b compares a specific case of $m = 64$ and original KAC. A speedup from $O(|S|^2)$ to $O((|S|/2^6)^2) = O(|S|^2/2^{12})$ is observed for $|S| \geq 2^{14}$.

6.1.2 Variable Chunking

In variable chunking where all GGM nodes are encrypted under KAC, the amount of KAC ciphertext increases from n to $2n - 1$. Hence, the storage required for `param` and $\mathfrak{C}$ will be twice of that of a normal KAC scheme (Sections 4.1.2 and 4.1.3). As every node is encrypted with

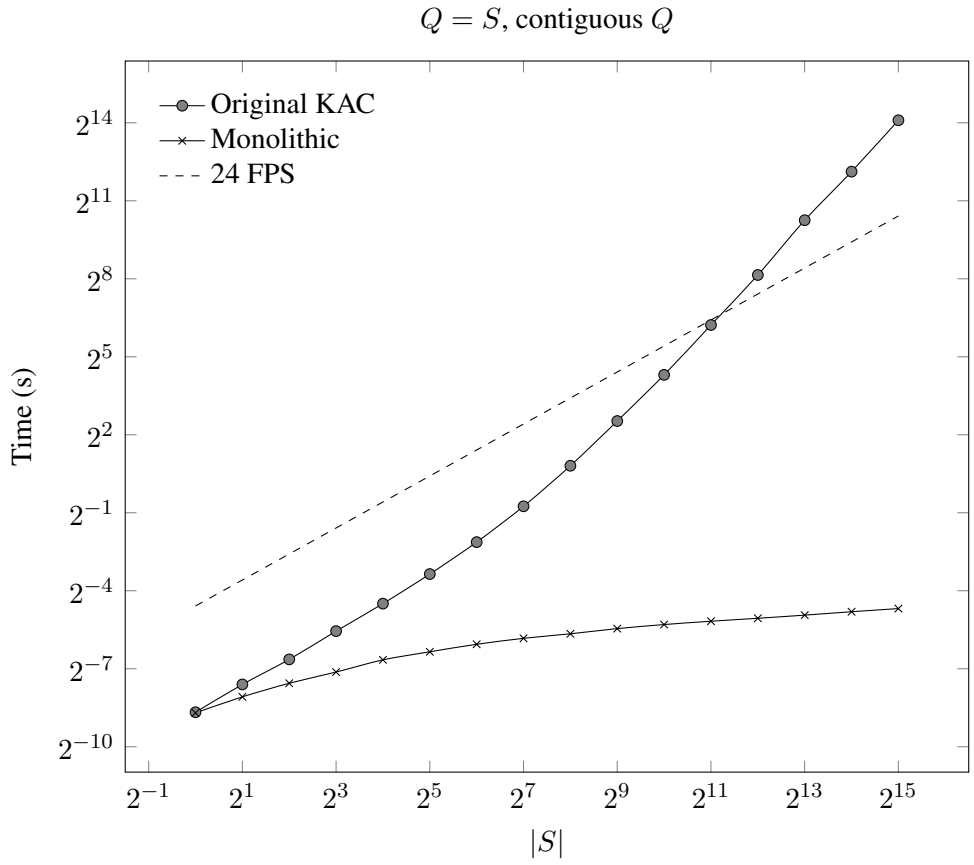


Figure 6.2: Monolithic KAC-GGM decryption time

KAC, KG is able to issue a constant-size KAC aggregate-key for any combination of nodes. Hence, the meta-key is always $O(1)$ in size. $O(\log(n))$ nodes is sufficient to represent any contiguous range. Therefore, using KAC's **Decrypt** algorithm together with variable chunking will require only $O(\log^2(n))$ decryption time at most. In our experiments shown in Fig. 6.2, variable chunking demonstrated short runtimes even for large datasets, much faster than realtime requirements of 24FPS. Variable chunking KAC-GGM hybrid scheme managed to generate all frame keys under 1 sec even for datasets as large as $|S| = 2^{14}$.

Despite the huge improvements in runtime, the significant increase in storage costs incurred by KG and RC is undesirable. Instead of encrypting all stratum in the GGM tree, DO may opt to skip and encrypt only specific strata in the tree. In the example shown in Fig. 6.3, nodes are encrypted once every 2 strata ($\phi = 2$).

Leaving gaps and encrypting once every ϕ strata will reduce the number of encrypted nodes from $2n-1$ to $(2^{\phi}\, n-1)/(2^{\phi}-1)$, reducing storage required in RC. However, this comes at the expense of decryption efficiency since more nodes now are required to represent a contiguous range of values. The number of nodes is needed to represent a single range is increased from $O(\log(n))$ to $O(2^{\phi} + \log(n))$.

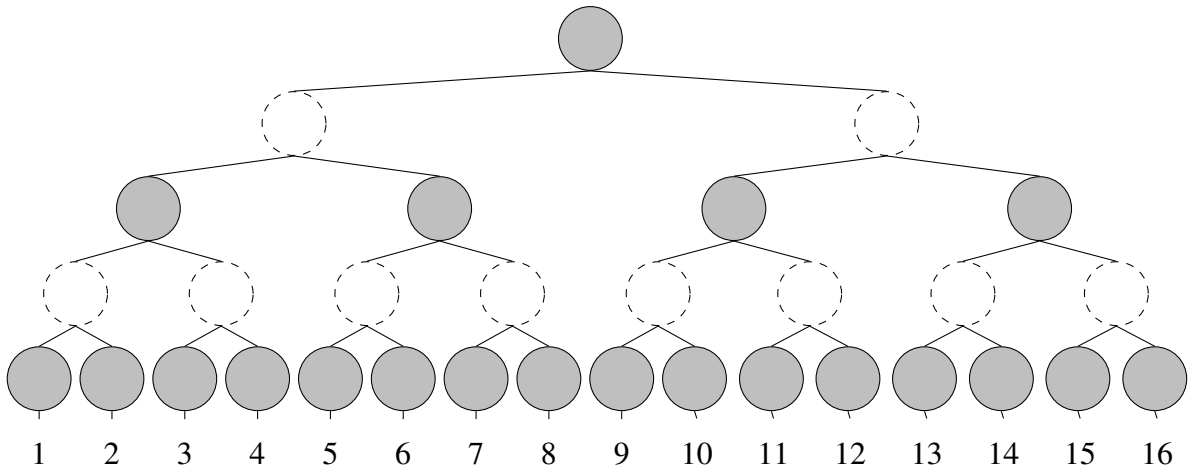


Figure 6.3: Generalized KAC-GGM with $\phi = 2$

6.2 KAC Decryption Optimization

In this section, we analyze the asymptotic and empirical runtimes of KAC decryption optimizations discussed in Section 5.2. A comparison of decryption complexities and number of operations needed for various algorithmic optimizations is presented in Table 6.2.

	Multiplication	Pairing	Division	Total
Unoptimized KAC	$\|Q\|(2\|S\|+1)$	$2\|Q\|$	$\|Q\|$	$2\|Q\|\|S\|+\Theta(\|Q\|)$
λ reuse	$\|Q\|(\|S\|+1)+\|S\|$	$2\|Q\|$	$\|Q\|$	$\|Q\|\|S\| + \Theta(\|Q\|) + \Theta\|S\|$
λ, ρ_i reuse	$3\|Q\| + 2\|S\| - 2$	$2\|Q\|$	$2\|Q\| - 1$	$7\|Q\| + 2\|S\| - O(1)$
KP-ABE	$\|Q\|O(\log(n))$	$\|Q\|O(\log(n))$	$\|Q\|O(\log(n))$	$\|Q\|O(\log(n))$

Table 6.2: Decryption complexity (Contiguous S, Q, arithmetic sequences of $d = 1$)

For arithmetic sequences, $Q = S$, reusing ρ_i results in asymptotic runtime improvement from $\Theta(|S|^2)$ to $\Theta(|S|)$. Experimental results in Fig. 6.4 showed that decryption for arithmetic sequences are much faster than KP-ABE as well as realtime requirements of 24FPS.

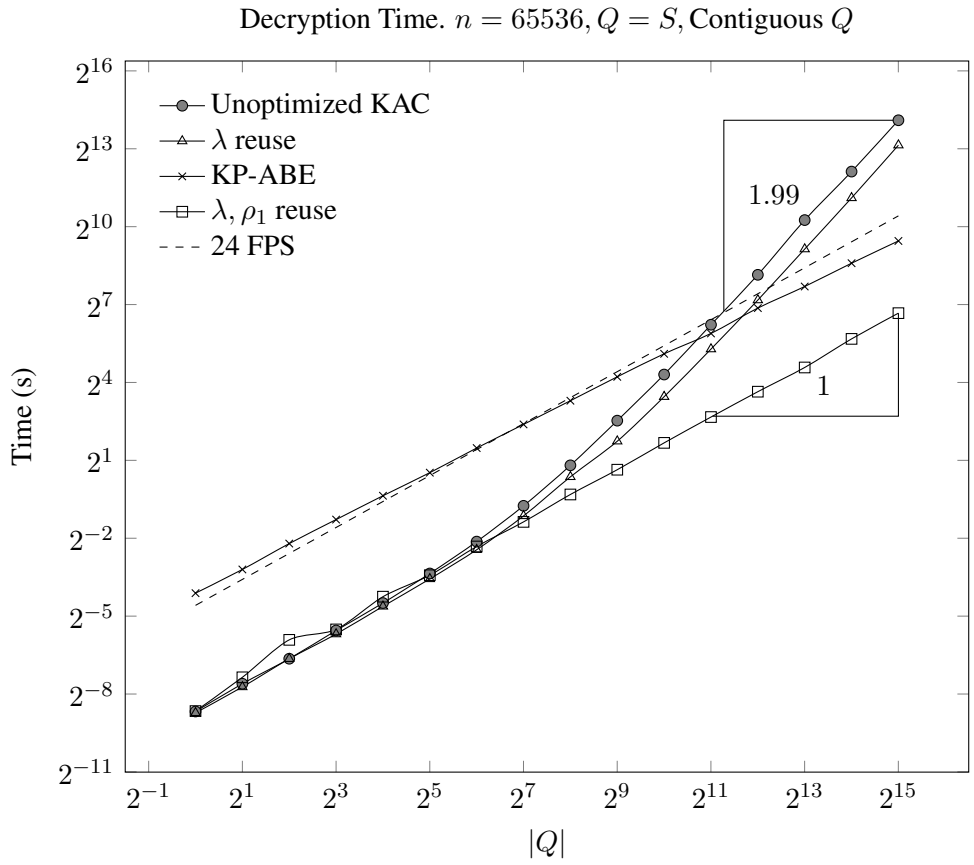


Figure 6.4: Optimized KAC (Speedup $= S^{1.99}/S^1 \approx S$)

6.3 Homomorphic MAC

We will examine the homomorphic MAC verification protocol described in Section 5.3 on both its security as well as its efficiency (storage and runtime).

6.3.1 Security

We prove that construction of the homomorphic scheme is information-theoretically secure against spoofing attacks even against an adversary capable of computing discrete logarithms.

The adversary has knowledge of the following public parameters which are obtained from RC.

$$\texttt{param} = \{g, g_1, g_2, \ldots g_n, g_{n+2}, \ldots, g_{2n}\} = \{g, g^{r_1}, g^{r_2}, \ldots g^{r_n}, g^{r_{n+2}}, \ldots, g^{r_{2n}}\}$$
$$\texttt{param\_sig} = \{a_1, \ldots, a_n\} = \{g_1^{\beta} g^{\beta_1}, \ldots, g_n^{\beta} g^{\beta_n}\}$$
$$\texttt{pk} = g^{\gamma}$$

We will assume that the adversary is capable of computing discrete logarithms efficiently. Hence, the adversary is able to obtain the following values:

$$r_1, r_2, \ldots, r_n, r_{n+2}, \ldots r_{2n} \tag{6.1}$$

$$\beta + r_1\beta_1,\ \beta + r_2\beta_2,\ \ldots,\ \beta + r_n\beta_n \tag{6.2}$$

$$\mathtt{msk} = \gamma \tag{6.3}$$

Combining 6.1 and 6.2, the adversary is able to obtain a system of linear equations $\mathbf{Bx} = \mathbf{C}$, where

$$\mathbf{B} = \begin{pmatrix} 1 & r_1 & 0 & 0 & \cdots & 0 \\ 1 & 0 & r_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & 0 & \cdots & r_n \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} \beta \\ \beta_1 \\ \vdots \\ \beta_n \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n + 1 \end{pmatrix} \tag{6.4}$$

Lemma 1. *The rank of matrix* $\mathbf{B}$ *is* n*.*

Proof. By the construction of **Setup** algorithm in KAC, $r_1, r_2, r_3, \ldots$ are all non-zero values. Reducing $\mathbf{B}$ to its row-echelon form, we obtain

$$\mathbf{B} \sim \begin{pmatrix} 1 & r_1 & 0 & \cdots & 0 & 0 \\ 0 & -r_1 & r_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -r_{n-1} & r_n \end{pmatrix}$$

Since the row-echelon form of $\mathbf{B}$ contains n non-zero rows, the rank of B is n.

Lemma 2. *If an adversary is able to forge a tuple* $\langle \tilde{w_1}, \tilde{w_2}, w_3 \rangle$*, where* $\tilde{w_1} = \prod_{i \in \tilde{w_3}} g_{n+1-i}$, $\tilde{w_2} = \tilde{w_1}^{\beta} \times \prod_{i \in w_3} g^{\beta_{n+1-i}}$, $\tilde{w_3} \neq w_3$*, then he/she is able to obtain* β*.*

Proof. Assume that the adversary is capable of computing discrete logarithms efficiently.

1. Applying **Computation** algorithm described in Section 5.3.1, the adversary computes a valid $\langle w_1, w_2, w_3 \rangle$ tuple.
2. Solving the discrete logarithm for $w_1,\ w_2$, the adversary obtains $R = \sum_{i \in w_3} r_i$ and $Y = \beta R + \sum_{i \in w_3} \beta_{n+1-i}$.

3. The adversary forges tuple $\langle \tilde{w_1}, \tilde{w_2}, w_3 \rangle$.

4. Solving discrete logarithms for $\tilde{w_1},\ \tilde{w_2}$, adversary obtains $\tilde{R} = \sum_{i \in \tilde{w_3}} r_i$ and $\tilde{Y} = \beta \tilde{R} + \sum_{i \in w_3} \beta_{n+1-i}$ from $\langle \tilde{w_1}, \tilde{w_2}, w_3 \rangle$

5. Since adversary knows the values for $Y, \tilde{Y}, R$ and $\tilde{R}$, he/she is able to obtain β by solving equations $\tilde{Y}$ and Y.

We assume that it is easy for an adversary to forge a tuple. As a consequence of Lemma 2, this implies that the adversary is able to compute β easily.

In Lemma 1, we proved that $rank(\mathbf{B}) = n$. Since there are $n + 1$ variables in $\mathbf{x}$, there is infinitely many solutions for β. Hence, it is extremely difficult for the adversary to compute β. This is a contradiction to the earlier statement that β can be easily computed. Therefore, we have established that it is difficult for an adversary to forge a valid tuple.

6.3.2 Efficiency

KG is able to use a hash function that outputs element of $\mathbb{Z}_p$ to generate $\beta_1, \ldots, \beta_n$. In our implementation, the hash function is defined as $\beta_i = H(\beta||i)$. With the protocol, it is no longer necessary to store `param` in KG. This reduces KG's storage needed from $\Theta(n)$ to $O(1)$. `param_sig`, a $\Theta(n)$ size public parameter is generated by **Signing** and stored in RC. Since RC already has to store $\Theta(n)$ ciphertext and `param`, the storage complexity remains unchanged.

Table 6.3 summarizes the various computations involved in the homomorphic MAC scheme. The most expensive operation is the generation of `param_sig`. It involves $\Theta(n)$ exponentiations, multiplications and hashes. However, since it is only performed once by DO during **Synthesis** phase (Section 3.2), it does not have an impact on the responsiveness of the system. For **Computation**, AU has to compute w_1, w_2, incuring a total of $2|S|$ multiplications. In **Verification**, KG has to do $|S|$ addition and hash operations and 3 exponentiations.

Parameter	Addition	Multiplication	Exponentiation	Hash	Algorithm
`param_sig`	0	n	$2n$	n	**Signing**
$w_1 = \prod_{i \in S} g_{n+1-i}$	0	$\|S\|$	0	0	**Computation**
$w_2 = \prod_{i \in S} a_{n+1-i}$	0	$\|S\|$	0	0	**Computation**
`aggregate_sig`	$\|S\|$	0	2	$\|S\|$	**Verification**
$K_S = w_1^{\beta}$	0	0	1	0	**Verification**
$K_S = \prod_{i \in S} g_{n+1-i}^{\gamma}$	0	$\|S\|$	1	0	**Extract** (Original KAC)

Table 6.3: Runtime complexity of the verification protocol

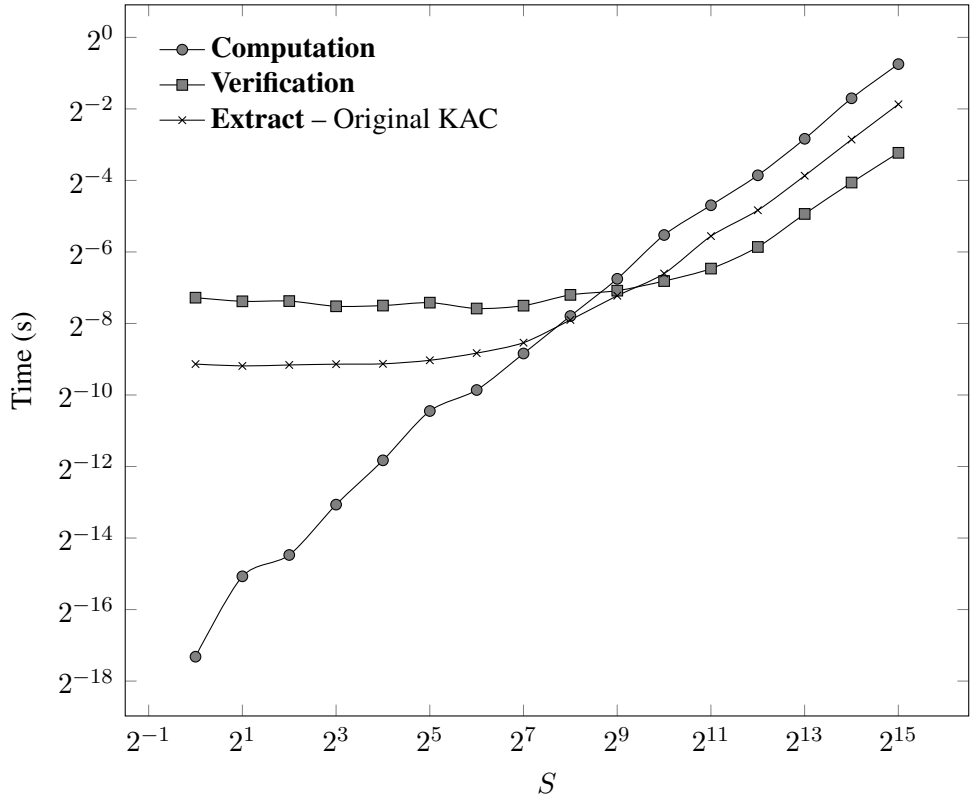


Figure 6.5: Homomorphic MAC K_G generation time

Fig. 6.5 illustrates some of the experimental results that we have obtained. **Computation** is performed solely by AU. We see that **Computation** can be performed under 1 sec even for large data sizes ($S = 2^{15}$).

The hashing and addition operations for **Verification** are performed in $\mathbb{Z}_p$. On the other hand, the multiplication operations for **Extract** algorithm in unmodified KAC are performed in bilinear group $\mathbb{G}$. As operations in $\mathbb{Z}_p$ are faster than operations in pairing group $\mathbb{G}$, **Verification** exhibits faster empirical runtime than KAC's **Extract** algorithm for large $|S|$ (i.e. $|S| > 2^{10}$).

6.4 Layered Symmetric Encryption

The layered scheme described in Section 5.4 requires additional intermediate layers, as discussed in Section 5.4. This incurs additional storage space of $(l-1)(n) = O(ln)$ in RC. This is in addition to l KAC-GGM structures generated for each content layer. Hence, the total storage in RC is $l\,O(2n-1) + (l-1)n = O(ln)$. Each layer requires $O(1)$ KG storage, or $O(l)$ KG storage for l layers. As symmetric decryption is used for all layers except the highest layer, decryption time is unlikely to be greatly impacted.

6.5 Summary

In this section, we present a summary of the entire scheme by applying our proposed system described in Chapter 5 to our security model discussed in Chapter 3.

6.5.1 Interactions

Synthesis

1. DO uses the KAC's Setup and KeyGen algorithm to generate `pk`, `msk`, and `param`. The video frames are encrypted using keys derived from Monolithic KAC-GGM scheme discussed in Section 5.1.2. DO then performs the **Signing** algorithm (Section 5.3.1) and outputs β and `param_sig`.

Upload

2. DO uploads `msk`, g and β to KG.
3. DO uploads the entire KAC-GGM structure, `param`, `param_sig` and the encrypted video frames to RC.

Retrieval

4. AU downloads the encrypted frames, `param` and `param_sig` from RC.
5. AU performs the **Computation** algorithm (Section 5.3.1) and forwards $\langle w_1, w_2, w_3 \rangle$ to KG.

Key Distribution

6. KG checks AU's access rights using an external verifier. Once AU is confirmed to have rights to access w_3, KG performs **Verification** (Section 5.3.1). If `aggregate_sig` is valid, KG generates K_{w_3}.
7. KG forwards K_{w_3} to AU.

8. Using `param`, K_{w_3} and the Monolithic KAC-GGM, AU is able to derive the relevant frame keys and use them to decrypt the encrypted video frames.

6.5.2 Performance Requirements

KG Bandwidth The bandwidth required between DO and KG is only $O(1)$ since only `msk` and β is transferred from DO to KG. Likewise, the bandwidth required between KG to AU is only $O(1)$ since K_{w_3} is constant-size, much lower than the desired bandwidth requirements of $O(\log(n))$.

KG Storage The storage required $O(1)$ since KG only stores `msk`, g and β, which is below the $O(\log(n))$ requirement.

DO Storage The monolithic KAC-GGM structure, `param`, `param_sig` and the encrypted video content all have a storage overhead of $\Theta(n)$ which is within our requirements.

Encryption/Decryption Time Generating all the parameters required and encrypting the video takes $O(n)$ time. Generating the frame keys takes $O(\log^2(n))$ for contiguous sets and $O(|S|)$ for arithmetic sequences. Empirical results in Fig. 6.2 showed that rate of frame key generation for large contiguous sets is much faster than realtime requirements of 24FPS.

6.5.3 Security Requirements

Unauthorized data access To gain encryption keys that is outside of the AU's access rights, the AU has to successfully forge the signature used in the verification protocol. In Section 6.3.1, we have proved that it is extremely difficult for an adversary to forge the signature.

Malicious/Spoofed RC The KAC-GGM structure stored in RC is encrypted using `msk` which is stored in KG. Both `param` and `param_sig` are public parameters. Therefore, RC is unable to gain any plaintext information.

KG Spoofing Since KG is likely to be a self-hosted server within the organization, we are able to use public key certificate to ascertain the identity of KG. As the private key of the certificate is only known to KG and the certificate is signed by a trusted authority, it is difficult for an adversary to impersonate as KG.

Collusion-resistant Our system uses the KAC's aggregate key as the meta-key. As discussed in Section 2.4, the aggregate key generated by KAC cryptosystem is collusion-resistant. Therefore, our system is collusion-resistant.

Chapter 7

Conclusion

As data-storage outsourcing becomes an increasingly common trend among organizations, client-side encryption is used to ensure data confidentiality. This raises issues of key inventory and distribution.

In this project, we have proposed an adaptation of KAC (Chu et al., 2013) which allows for effective management of encryption keys for time-series data, such as video surveillance data. Our proposed scheme requires $O(1)$ private storage through the use of a homomorphic signature scheme, $O(1)$ bandwidth costs for key distribution through the use of KAC's aggregate key, and $O(\log^2(n))$ time for key generation for a contiguous range of data points using a KAC-GGM hybrid scheme. The scheme also supports fast key generation for layered content and varying video frame rates. Since our system does not require any form of computation by the cloud server, the scheme can be deployed over existing cloud providers without any constraints. Moreover, **KG** performs only a small proportion of the computation. The bulk of the computation required for key generation is performed by the users themselves. Hence, the system is highly scalable and is able to support a large number of users.

One limitation to our scheme is the slow decryption speed for non-contiguous, random data points. For such data sets, decryption speed degrades to the unmodified KAC algorithm.

Apart from single dimensional time-series data, our work could be extended to support arbitrary attributes (e.g. GPS coordinates) which spans multiple dimensions. To support such functionalities, existing techniques such as MRQED and ABE will have to be explored and integrated.

References

Akinyele, J. A., Garman, C., Miers, I., Pagano, M. W., Rushanan, M., Green, M., & Rubin, A. D. (2013).
Charm: a framework for rapidly prototyping cryptosystems.
Journal of Cryptographic Engineering, *3*(2), 2013, 111–128.

Bethencourt, J., h. Hubert Chan, T., Perrig, A., Shi, E., & Song, D. (2006).
Anonymous multiattribute encryption with range query and conditional decryption.
Security and Privacy, 2006. SP'06. IEEE Symposium on, 2006.

Bethencourt, J., Sahai, A., & Waters, B. (2007).
Ciphertext-policy attribute-based encryption.
Security and Privacy, 2007. SP'07. IEEE Symposium on (pp. 321–334), IEEE, 2007.

Chu, C., Chow, S., Tzeng, W., Zhou, J., & Deng, R. (2013).
Key-aggregate cryptosystem for scalable data sharing in cloud storage.
Parallel and Distributed Systems, IEEE Transactions on, Vol. PP (pp. 1–1), 2013.

Goldreich, O., Goldwasser, S., & Micali, S. (1986).
How to construct random functions.
Vol. 33 (pp. 792–807), New York, NY, USA, August, 1986: ACM.

Goyal, V., Pandey, O., Sahai, A., & Waters, B. (2006).
Attribute-based encryption for fine-grained access control of encrypted data.
Proceedings of the 13th ACM conference on Computer and communications security (pp. 89–98), ACM, 2006.

Kamara, S., & Lauter, K. (2010).
Cryptographic cloud storage.
In *Financial cryptography and data security* (pp. 136–149).

Lewko, A., Sahai, A., & Waters, B. (2010).
Revocation systems with very small private keys.
Security and Privacy, 2010. SP'10 IEEE Symposium on (pp. 273–285), IEEE, 2010.

Ostrovsky, R., Sahai, A., & Waters, B. (2007).
Attribute-based encryption with non-monotonic access structures.
Proceedings of the 14th ACM conference on Computer and communications security (pp. 195–203), ACM, 2007.

Sahai, A., & Waters, B. (2005).
Fuzzy identity-based encryption.
In *Advances in cryptology–eurocrypt 2005* (pp. 457–473).

Shi, E., Bethencourt, J., Chan, T.-H., Song, D., & Perrig, A. (2007).
Multi-dimensional range query over encrypted data.
Security and Privacy, 2007. SP'07. IEEE Symposium on (pp. 350–364), IEEE, 2007.

Waters, B. (2011).
Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization.
Public Key Cryptography–PKC (pp. 53–70), 2011.

Wu, Y., Wei, Z., & Deng, R. (2013).
Attribute-based access to scalable media in cloud-assisted content sharing networks.
Multimedia, IEEE Transactions on, Vol. 15 (pp. 778–788), 2013.