Josef Edgar Sabongui, BSc

Usable Attribute-Based Encryption For Industrial Cloud Systems

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors

Dipl.-Ing Herbert Leitold and Dipl.-Ing. Dominik Ziegler, BSc

Evaluator

O.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch

Institute of Applied Information Processing and Communications
Head: O.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch

Graz, May 2019

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

Attribute-Based Encryption allows for encryption of data according to a policy, which has to be satisfied successfully to decrypt a ciphertext. Based on bilinear pairings, these cryptosystems are a relatively novel field of research. While providing new ways to design cryptographic schemes, inherent challenges like key escrow or computational performance issues have to be addressed when adopting these systems.

We examine the requirements that an Attribute-Based Encryption system would need to fulfill in the scenario of the Industrial Internet of Things. This environment is comprised of a heterogeneous group of components, ranging from computationally potent cloud services to severely constrained sensors. Furthermore, data confidentiality is a pivotal aspect to consider when information is shared throughout such a vast environment.

We present a system that is built around data access control and tries to adapt to the strength and weaknesses of the involved actors. Our system mitigates key escrow through keys that are split among services and endpoints. To relieve potentially resource limited endpoints from performing computationally expensive pairing operations, partial decryption is provided by a dedicated service. We give a detailed mathematical description of the architecture and present a prototype that implements all roles in the system. This thesis shows that adopting Attribute-Based Encryption in the Industrial Internet of Things is feasible in regards to performance and provides data access control on a cryptographic level.

Kurzfassung

Attribute-Based Encryption ermöglicht die Verschlüsselung von Daten anhand von Regeln die erfüllt werden müssen um einen Chiffretext erfolgreich entschlüsseln zu können. Diese Kryptosysteme, basierend auf bilinearen Paarung, sind ein relativ junges Forschungsgebiet. Während sich damit neuartige kryptographische Schemata entwerfen lassen, müssen unmittelbar verbundene Herausforderungen wie Key Escrow oder Berechnungsgeschwindigkeit bei einem Systemeinsatz bedacht werden.

Wir untersuchen anhand des Industrial Internet of Things Szenarios welche Anforderungen ein Attribute-Based Encryption System erfüllen muss. Diese Umgebung besteht aus einer heterogenen Gruppe von Komponenten, beginnend bei leistungsstarken Cloud Diensten bis hin zu stark eingeschränkten Sensoren. Desweiteren is die Vertraulichkeit von Daten ein entscheidender Aspekt wenn Informationen in einer derartig weitläufigen Umgebung geteilt werden.

Wir präsentieren ein System, das Vertraulichkeit von Daten zum Kern hat und versucht, die Stärken und Schwächen aller involvierten Akteure zu berücksichtigen. Unser System entschärft Key Escrow indem Schlüssel auf mehrere Services und Endpunkte aufgeteilt werden. Ressourcenmäßig eingeschränkten Endpunkten werden aufwändige Berechnungen von Paarungen erspart, da eine partielle Enschlüsselung durch ein dediziertes Service erfolgt. Wir stellen eine detaillierte mathematische Beschreibung der Architektur zur Verfügung und präsentieren einen Prototypen der alle Rollen im System implementiert. Wir zeigen, dass Attribute-Based Encryption im Industrial Internet of Things performant umsetzbar ist und Zugriffskontrolle auf kryptographischer Ebene gewährleistet.

Contents

1. Introduction

The so called *Industry 4.0* [35] is an idea that has its roots in an initiative from the German government. The vision behind it being, that traditional production processes would greatly benefit from being enhanced with modern information technology. But information driven production and optimization requires massive amounts of data throughout entire production lines. Due to the ever increasing complexity of these systems and factories as a whole, data needs to be gathered and analyzed in every step of a production process. Shrouf, Ordieres, and Miragliotta [53] argue that energy efficiency of factories could be optimized by smart meters installed throughout factories. Zhou, Liu, and Zhou [64] recommend to utilize big data analysis to optimize manufacturing processes and reduce costs. Therefore, it is desired to equip all types of machinery with sensors and actuators. This includes stationary equipment, autonomously moving robots as well as equipment of factory workers. These devices represent nodes in an interconnected environment. This is a industry focused variant of Internet of Things (IoT), the so called Industrial Internet of Things (IIoT).

Aside from production lines, the entire manufacturing facility can be seen as a pool of information. Be it information about its warehouse, temperature and humidity throughout the production plant, energy metrics up to heating and air condition systems. All this data can be used to improve production efficiency. The value of this information is not limited to the facility operators. Analyzing this data can also benefit other departments of a production company, their customers or external partners like maintenance contractors.

Due to the heterogeneous nature of this environment, the IIoT represents an approach to collect data throughout all these systems. This plethora of information needs to be collected, stored, distributed and analyzed. Some

devices are limited in storage capacity and computational power, therefore offloading storage and analysis tasks to more potent systems seems like a reasonable strategy, as Samie, Bauer, and Henkel [50] discuss in the context of IoT in the healthcare sector. Cloud based solutions appear to fulfill the requirements of a scenario where a significant set of participating devices operates with limited resources. They offer high availability as well as massive computing power and storage capabilities. There already exist cloud based IIoT services like *Siemens MindSphere* [54], *Predix*[45] and *IIoT by Honeywell* [32].

Such an ecosystem inherently faces challenges regarding security and confidentiality. It is undesirable to unconditionally share this wealth of information with other entities. Confidentiality of data is a crucial requirement for the IoT in an enterprise setting, as Sadeghi, Wachsmann, and Waidner [48] point out. Therefore, fine-grained data access control is indispensable in an environment like this. In the setting of cloud systems, Cloud Service Providers (CSPs) have control over all data hosted in and processed with their services. This lack of authority can constitute an obstacle for companies when it comes to adopting said systems.

Traditionally, data has been secured by encrypting it and exchanging the accompanying key. Symmetric algorithms like Advanced Encryption Standard (AES) [13] allow for very efficient and fast encryption and decryption of data. Public-Key cryptosystems like Rivest–Shamir–Adleman (RSA) [47] are built around key pairs, where a public key is used to encrypt data that can only be decrypted by an accompanying secret key. In both scenarios, an encryption is tied to a specific key or key-pair.

Attribute-Based Encryption (ABE) moves away from this paradigm and specifies access policies that need to be fulfilled in order to decrypt a ciphertext. Based on the chosen approach, such a policy is either attached to keys or ciphertexts, called Key-Policy Attribute-Based Encryption (KP-ABE) [24] and Ciphertext-Policy Attribute-Based Encryption (CP-ABE) [6] respectively. In a CP-ABE setting, a Data Owner (DO) encrypts data under a given access policy. This resulting ciphertext can only be decrypted by keys which are defined by attributes that satisfy the policy. This allows for cryptographically ensured Attribute-Based Access Control (ABAC) [28]. The

promise that data access is enforced on this level, can be a step towards improving trust in the confidentiality of data in IIoT systems.

But ABE is subject to several challenges. Bilinear pairings build the foundation in ABE designs. Depending on the architecture of the cryptosystem and the chosen pairings, computational requirements can exceed capabilities of more resource constrained devices typically found in IIoT environments. Private keys are always created by some sort of Key Authority (KA), which raises concerns regarding privacy of keys and confidentiality of data. Another important aspect is the implementation of key revocation, which is a requirement for all systems that implement access control.

1.1. Contribution

The aim of this thesis is to demonstrate that ABE can be a suitable tool for ensuring fine-grained access control and thus providing data confidentiality in IIoT environments. We build upon the work of Lin, Hong, and Sun [38], which addresses various problems an ABE system would face in an IIoT environment. Their architecture was defined in the Type I setting of bilinear pairings. These symmetric pairings are prone to security issues and bad performance. Therefore, we adapt the architecture to work in the more efficient and secure Type III setting. We provide a prototype of the adapted system and evaluate it with regards to ciphertext overhead and execution time of operations.

1.2. Structure

We will present the fundamental building blocks of our chosen ABE system in Chapter 2. As ABE is usually built on Pairing-Based Cryptography (PBC), we will discuss bilinear pairings and access structures. Furthermore, we will give an overview to Multi-Party Computation (MPC) as it is essential to the architecture of our prototype. In Chapter 3 we will discuss the history of ABE, problems inherent to it, as well as approaches to mitigate those challenges. Chapter 4 presents the mathematical foundation of our ABE

system. The implemented prototype and component descriptions will be provided in Chapter 5. At last, we will discuss the results from evaluating our implementation in Chapter 6.

2. Background

The term ABE is used to describe a cryptosystem in which users are able to decrypt ciphertext if a defined policy is fulfilled. These systems are usually built upon bilinear pairings. Furthermore, they rely on access structures to model the policy under which data is secured. ABE systems usually come in two variants. KP-ABE assigns policies to keys and attributes to ciphertexts, while CP-ABE does the opposite. More detailed information on different ABE approaches will be discussed in Chapter 3. As will be shown in Chapter 4, the proposed system also relies on the secure computation of certain values through MPC.

This chapter explores the most significant techniques that were required to implement the thesis' system prototype.

2.1. Bilinear Pairings

Bilinear pairings first rose to prominence in cryptography, when they were used to improve attacks on the Elliptic Curve Discrete Logarithm Problem (ECDLP) by Menezes, Okamoto, and Vanstone [40]. They demonstrated how to reduce the problem of ECDLP of supersingular curves to the Discrete Logarithm Problem (DLP) of finite fields, for which subexponential attacks were already known at the time. They employed the Weil Pairing to map points from the elliptic curves to an extension of the underlying finite field, where they then calculated the discrete logarithm using index-calculus methods. This method is usually abbreviated MOV. The first utilization of bilinear pairings for constructive use was presented by Joux [34], where he demonstrated a Diffie-Hellman (DH)-like protocol in which 3 users can calculate a common secret in only one round.

2.1.1. General

A pairing e maps points in two abelian groups $\mathbb{G}_1$, $\mathbb{G}_2$ of the same order p to a target group $\mathbb{G}_T$. As pairings are usually based on Elliptic Curves (ECs) we use the notion of additive groups for $\mathbb{G}_1$ and $\mathbb{G}_2$, while $\mathbb{G}_T$ is presented as a multiplicative group.

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T \tag{2.1}$$

With group generators P_1, P_2, bilinear pairings require linearity in both arguments:

$$e\left(aP_1, bP_2\right) = e\left(P_1, P_2\right)^{ab} \tag{2.2}$$

Pairings also have to be non-degenerate, which means that the generators must not map to the identity element of $\mathbb{G}_T$:

$$e\left(P_1, P_2\right) \neq 1 \tag{2.3}$$

And lastly, pairings have to be efficiently computable. Such a polynomial time algorithm was first described by Miller [41].

2.1.2. Elliptic Curves

In practice, pairings are implemented using ECs, which are defined over some finite fields $\mathbb{F}_q$. For detailed mathematical descriptions of ECs, we refer to the book on EC arithmetic by Silverman [55]. Among other specific curve properties, he describes the two main groups in which ECs are categorized, namely supersingular and ordinary curves. An additional important aspect of ECs to consider is their embedding degree k. The ECDLP can be reduced to the DLP of a field extension $\mathbb{F}_{q^k}$, by employing the previously mentioned MOV algorithm.

Supersingular Curves For supersingular curves, it was shown by Menezes, Okamoto, and Vanstone [40] that the embedding degree is $k \leq 6$. Due to improving attacks on curves with small embedding degrees, these curves should be avoided when implementing PBC systems. Examples of breaking these curves in practice were demonstrated by Granger, Kleinjung, and Zumbrägel [26] as well as Adj, Menezes, and Oliveira [1].

Ordinary Curves Miyaji, Nakabayashi, and Takano [42] gave the parameters to construct curves with prime order and embedding degrees up to $k = 4$, which are referred to as MNT curves. Page, Smart, and Vercauteren [44] compared several MNT and supersingular curves regarding security and efficiency. They showed that MNT variants required smaller ECs to achieve comparable security, while the efficiency depended on the scheme to be implemented. In 2006, Barreto and Naehrig [2] presented a way to construct pairing friendly prime order curves with an embedding degree $k = 12$. These so-called Barreto-Naehrig (BN) curves exhibit a strong security level, even when choosing relatively small ECs. They also showed how to heavily compress the representation of points and pairings for these curves, resulting in lower bandwidth requirements.

2.1.3. Pairings

Beside the Weil method, which was used in early PBC systems, there exist other approaches to create bilinear pairings. The Tate pairing was initially used in a cryptographic context to tackle the ECDLP by Frey and Ruck [20] and furthermore showed to be more efficient than the Weil pairing, as was demonstrated by Galbraith, Harrison, and Soldera [21]. In 2005 Barreto et al. [3] first presented their Eta pairing, a Tate variant that exhibited improved efficiency on supersingular curves. The Eta pairing was then modified into the so called Ate pairing by Hess, Smart, and Vercauteren [27] to work in the setting of ordinary curves, with additional speedup of the pairing operation. In 2010 Vercauteren [59] presented a method to calculate optimal Ate pairings for several families of pairing friendly ECs.

2.1.4. Pairing Types

Bilinear pairings can be categorized into 4 main types, each with distinct properties. One defining property is the existence of an efficiently computable group isomorphism ψ between the paired groups $\mathbb{G}_1$ and $\mathbb{G}_2$. Some ABE systems require this isomorphism for either the constructions or security proofs. A detailed analysis of the use of ψ in ABE protocols was given by Chatterjee and Menezes [9], where they also argue that ψ is not necessarily required for the functionality or security of most protocols but instead more likely an artifact from the initial research of PBC systems. The following classes have been identified so far:

- **Type I**: This type is unique in that $\mathbb{G}_1 = \mathbb{G}_2$, with the groups being supersingular curves. This leads to a clear isomorphism $\psi\colon \mathbb{G}_2 \to \mathbb{G}_1$ for these pairings. Such pairings are called symmetric pairings. Due to the previously mentioned characteristics of supersingular curves, these pairings are regarded as insecure or at least inefficient, as is discussed by Uzunkol and Kiraz [58].
- **Type II**: Contrary to Type I, these pairings are based on ordinary curves with $\mathbb{G}_1 \neq \mathbb{G}_2$. They also provide an efficiently computable isomorphism $\psi\colon \mathbb{G}_2 \to \mathbb{G}_1$. Due to the group inequivalence, the pairings are called asymmetric pairings.
- **Type III**: These are similar to Type II pairings, except for the absence of an efficiently computable group isomorphism. They do offer better performance than their Type II counterparts, as was shown by Galbraith, Paterson, and Smart [22].
- **Type IV**: Pairings of this type have a $\mathbb{G}_2$ with a different order than $\mathbb{G}_1$. But they do provide an efficiently computable homomorphism $\psi\colon \mathbb{G}_2 \to \mathbb{G}_1$. The first to classify this type were Chen, Cheng, and Smart [10] who based the definition on the thesis of Shacham [51].

2.2. Access Structures

ABE systems require some type of access structure to be able to enforce a policy on its ciphertexts and keys. As ABE systems can differ greatly

in their construction, the methods to encrypt and decrypt ciphertexts are intertwined with the choice of the access structure used.

2.2.1. Access Trees

When Goyal et al. [24] first presented their KP-ABE system, they opted for a tree access structure to define the policy associated with private keys. They employed a recursive algorithm for the decryption process, which traverses a given access tree to recover a plaintext. Similar approaches can be seen in proposals by Bethencourt, Sahai, and Waters [6], Ibraimi et al. [31] or Hur [29].

2.2.2. Linear Secret Sharing Schemes and Monotone Span Programs

Even though Goyal et al. [24] used a recursive decryption function, they showed that a construction utilizing Linear Secret Sharing Schemes (LSSSs) was possible. In a LSSS, a secret is distributed among multiple participants according to an access structure. Only sets of participants that fulfill the access structure are able to reconstruct the original secret with a linear combination of their pieces. Beimel [5] demonstrated that LSSSs are equivalent to Monotone Span Programs (MSPs), which were introduced by Karchmer and Wigderson [36]. A MSP is a matrix $\mathcal{M}$, where each row is labeled with a literal. In the setting of LSSS the labels would correspond to the participants of the scheme, while the rows correspond to the respective pieces. A MSP only accepts a subset of rows if they are a linear combination of a fixed nonzero vector. This is equivalent to fulfilling the access structure of a LSSS. A simple method to convert a boolean formula into an LSSS matrix was provided by Lewko and Waters [37]. A more advanced method that also supports threshold gates was presented by Liu and Cao [39].

Sharing With LSSS, one can distribute a secret $s \in \mathbb{Z}_p{}^*$ to n participants. To do this, a fixed non-zero vector has to be used for creating the share

matrix. Usually, this is either $(1, 1, \dots 1)$ or $(1, 0, \dots 0)$. Using one of the previously mentioned approaches, one ends up with a share matrix $\mathcal{M}$ with n rows. Each row is labeled, indicating to which participant it corresponds. This labeling is denoted as ρ. Subsequently, a vector $v = (s, x_1, x_2, \dots x_n)$ is created. To mask the secret value s in the next steps, all other values are chosen randomly. The final shares are generated through the following multiplication :

$$v_s = \mathcal{M} \cdot v^T = \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{pmatrix} \tag{2.4}$$

According to the labeling of $\mathcal{M}$, the resulting row entries of v_s are distributed to their respective participants according to ρ .

Recovery To successfully recover a secret s shared by an LSSS, one has to combine a set of authorized shares using factors that enable the reconstruction of s.

$$s = \sum_{t|\rho(t) \in \mathcal{S}} \omega_t \cdot \lambda_{\rho(t)} \tag{2.5}$$

These factors can only be recovered, if the set of participants $\mathcal{S}$ fulfills the access policy. The recovery can be achieved by solving a system of linear equations, based on a submatrix of $\mathcal{M}$ according to the set of participants $\mathcal{S}$. Examples for sharing and recovery of factors can be found in Appendix A.2.

2.3. Multi-Party Computation

The task to compute a result based on mutually secret inputs between two or more actors is called Multi-Party Computation (MPC). It is essential that no

party learns any other secret than its own while performing the computation. A prominent example of such a computation in the setting of two players is the so called *Millionaires' Problem*, where two parties want to determine who is richer without revealing their wealth to each other. Yao [62] gave possible solutions to this problem using one-way functions. In 1986, Yao [61] presented a general way to achieve Two-Party Computation (2PC). This would later become known as *Yao's Garbled Circuits*. Here, a boolean circuit that represents the desired computation function is obfuscated or 'garbled' by one of the two players. The second player then evaluates the garbled circuit using their own input and the encrypted inputs of the first player. After evaluation, both players gain knowledge of the result without ever learning the other player's secret input. A similar approach for the MPC setting has been presented by Goldreich, Micali, and Wigderson [23].

2.3.1. SPDZ

Damgård et al. [11] proposed a rather efficient MPC scheme using *Somewhat Homomorphic Encryption*, which was later named SPDZ based on the authors' names. The protocol is split into two main parts, an offline (or preprocessing) and an online phase. Independent of the actual inputs and function to be evaluated, the preprocessing stage prepares information that can be used in the second phase. This leads to much faster evaluation by the players during the online phase. Aside from its performance, the protocol stays secure against n-1 corrupt parties (with n being the total amount of players). The authors adapted the protocol[12] by moving more operations to the offline phase and providing improvements on an implementation level of the protocol.

3. Related Work

Research in the area of ABE has become increasingly popular in the past few years. While nearly all proposals share the foundation of bilinear pairings, different approaches to the subject have resulted in a variety of systems. This section outlines the history of ABE as well as inherent problems and proposed solutions.

Identity-Based Encryption In 1985 Shamir [52] first introduced the concept for a signature scheme that did not rely on generating random public/-private key pairs, but instead derived them from information about a user. In this scheme the public key is essentially the combination of information that uniquely identifies a user, while the secret key is then derived from this identity through trusted key generation centers. While the initial implementation only examined signatures in this identity-based system, Shamir assumed that identity-based cryptosystems would exist as well.

Boneh and Franklin [7] were the first to propose a fully functional Identity-Based Encryption (IBE) system based on the Weil pairing of ECs. Although they used the Weil pairing, the authors mentioned that their system could be used with any pairing, such as the Tate pairing for performance reasons, as long as certain DH assumptions hold. They furthermore showed how to equip this system with key escrow capabilities. A feature explicitly avoided in this thesis, as the key generating entity could maliciously recreate keys.

It was not until Sahai and Waters [49] published their Fuzzy Identity-Based Encryption (Fuzzy IBE) system, that research shifted towards systems that enabled users to decrypt ciphertexts if their private key was defined by more than just an exact identity match. The Fuzzy IBE system allowed decryption if an identity was considered 'close' to a target identity. This property was achieved through defining attributes and a tolerance factor determining

the allowed distance from the target identity. The main application of this IBE variant was seen in biometrics-based IBE systems due to the provided error-tolerance. However, they also first coined the term Attribute-Based Encryption and the potential application of it. While their approach worked well in the given context of biometric identities, the expressibility of access policies was limited by the simple threshold approach.

Attribute-Based Encryption Goyal et al. [24] created a cryptosystem they named KP-ABE in 2006. Their system equipped private keys with policies, which defined what ciphertexts users were able to decrypt. Here, the access structure was incorporated in the key while ciphertexts were labeled with attributes. This allowed for far more expressive access policies than the previously mentioned Fuzzy IBE system, like complex boolean formulas represented through access trees. However, a user that encrypts data can only equip the ciphertext with attributes and therefore has limited control over who is able to decrypt it, since the policies are assigned to keys by a key generating authority.

In 2007 Bethencourt, Sahai, and Waters [6] presented a system where policies would be attached to ciphertexts instead of keys, as previously shown by Goyal et al. They named this system CP-ABE. Here, the userkeys would be defined by attributes. Using this approach, data owners would be empowered to attach a detailed specification of a decryption key's attributes to the ciphertext. Compared to KP-ABE, the proposed system was more closely related to the principle of Role-Based Access Control (RBAC) [17] which is commonly employed throughout general access control systems.

Revocation In any system that assigns permissions to its users, it is necessary to have a strategy in place for revocation of aforementioned permissions. Early works in the field of ABE (and IBE) relied on periodically issuing keys to users in a system, as can be seen in Boneh and Franklin [7]. Expanding on that principle idea, Bethencourt, Sahai, and Waters [6] suggested that instead of just using a calender year as an attribute for the key's validity, a system could equip keys with concrete expiration dates by extending their attributes checks to support integer comparisons.

In a scenario based on health records, such revocation approaches seemed impractical to Ibraimi et al. [31]. They proposed a system they named Mediated Ciphertext-Policy Attribute-Based Encryption (mCP-ABE), that allowed for immediate revocation through the use of an additional authority in the system called a 'mediator'. In this environment a user is never in possession of the entire private key. Instead, the mediator always holds a part of a user's key and generates a message specific user token that is necessary to decrypt a ciphertext. If a user loses an attribute, the mediator is informed about the changed set of attributes and rejects any request where the access policy would require said attribute.

Hur [29] presented an ABE setup, where ciphertexts are re-encrypted by a storage server using re-encryption keys corresponding to attributes and users associated with them, so called attribute groups. If a user loses an attribute and drops out of an attribute group, the storage server creates a new re-encryption key for that particular attribute so that the ciphertext can only be decrypted by users satisfying the attribute requirements.

Key Escrow Mitigation Most of the previously discussed ABE systems are built around a single Key Authority that issues private keys derived from a masterkey. Therefore, KAs are generally capable of freely recreating every distributed key. Subsequently, a KA can decrypt every ciphertext created in a system. While this feature may be desired by some actors like state authorities, it also makes a KA the single point of failure in an ABE system.

Chase and Chow [8] developed a multi-authority ABE scheme where the masterkey is split among several attribute authorities, each managing a different set of attributes. In this system, users request decryption keys from the numerous authorities corresponding to the attributes they are entitled to. To decrypt a ciphertext they would have to be in possession of decryption keys from all responsible authorities managing the access policy's attributes.

Lewko and Waters [37] proposed to completely decentralize ABE, by letting every user in a system act as an ABE authority and issue keys for other users.

Hur [29] splits the responsibility of key generation to a key generation center and a storage server using secure 2PC. These two authorities craft a user's key using their respective secret keys, thus effectively preventing them from (re-)creating private keys on their own.

Wang et al. [60] presented a so called Accountable Authority Attribute-Based Encryption (A-ABE) system, where keys can be categorized into so called 'families'. Users are involved in the generation of keys, whose input determines the family their keys belong to. The key family is however unbeknownst to the key generation authority during generation. An authority can not definitively recreate a user's key in the same family. Therefore, detection of a key recreated by the authority alone is possible.

Efficiency In environments with restricted computational power, like the IoT, it is necessary to develop protocols that can be implemented in these settings. The calculation of pairings constitutes the most expensive part of ABE schemes. This is why both [24] and [6] already provided optimizations to their system's decryption process by reducing the amount of relevant attribute nodes of an access structure, for which pairings would actually have to be computed.

While most developed schemes try to reduce the use of pairings, there exist approaches to eliminate pairings all together. Yao, Chen, and Tian [63] created a system free of pairings especially targeted towards IoT devices, by solely constructing their scheme using EC primitives.

Another approach is to offload expensive parts of the computation from users and devices to dedicated servers, as can be seen in the work of Lin, Hong, and Sun [38]. Their system is based on Hur's architecture [29] but add another actor, a so called Decryption Server (DS) that partially decrypts ciphertexts for users. The users then only have to do one non-pairing based decryption step with their private key to gain access to the plaintext, therefore being spared of most of the expensive calculations in the system.

4. Architecture

We gave an overview to the requirements of IIoT in Chapter 1. We also discussed various problems inherent to Attribute-Based Encryption in Chapter 3. One of the main requirements we focused on, is the separation of power when using a cloud based system. A production oriented company has a vital interest in having control over who gets access to data generated at their facilities, while also profiting from the advantages cloud based services offer. The other major requirement is a result of the computational limitations IIoT devices may be subjected to. It is therefore essential, that the computational burden of adopting ABE in these environments is not overwhelming.

The work of Lin, Hong, and Sun [38] addressed several of these challenges and was therefore chosen to serve as the foundation of the proposed system, with respect to the ABE framework. A Client's private key is split three-ways, where Key Authority, Re-Encryption Server and the Client each hold their key-part. To fully decrypt ciphertexts, all these parts have to be used. Since the majority of decryption calculations are performed by the Decryption Server, no expensive bilinear pairings have to be done on the client's side. Furthermore, the servers' key-parts have to be retrieved on each decryption request. This ensures that key revocation is enforced instantaneously through the KA, by simply invalidating and updating user key-parts based on the change in their associated attributes. The split private keys also prevent key escrow, since no actor has knowledge of all three key-parts throughout the entire decryption process.

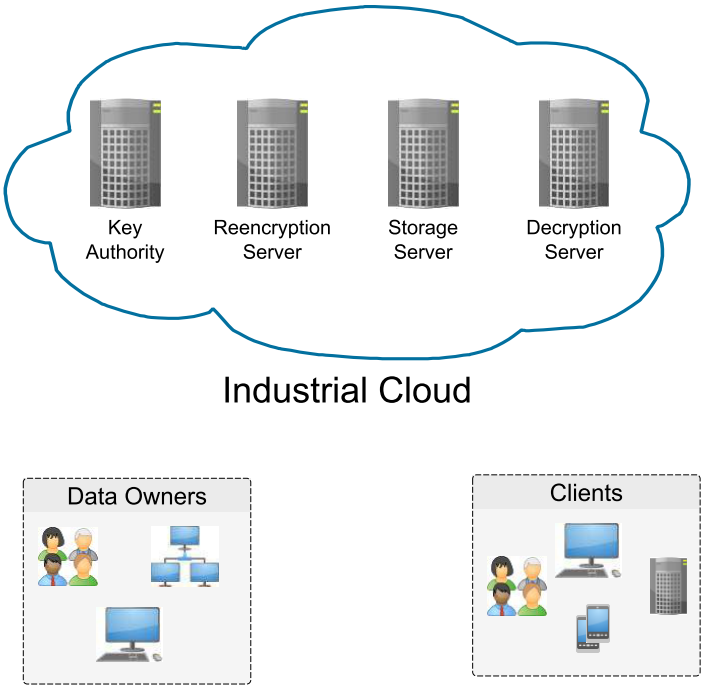

Figure 4.1.: Overview of ABE system actors

4.1. Actors

The architecture of this ABE system identifies several actors (see Figure 4.1), as outlined throughout this section. Each actor plays a significant role in the environment of this ABE system.

4.1.1. Key Authority

The Key Authority (KA) is in charge of key management in the system. It creates and updates client key-parts in conjunction with the Re-Encryption Server (RS). However, it remains the sole authority to grant or revoke attributes from aforementioned key-parts. Any modifications to the system parameters, such as additional attributes over the course of the systems lifetime, are exclusively performed by the KA. The KA is responsible to

notify other actors, such as the RS and DS, of any such system relevant changes.

4.1.2. Re-Encryption Server

The main responsibility of the Re-Encryption Server (RS) is re-encryption of initial ciphertexts, which are created and uploaded by data owners. The re-encryption process incorporates the attribute groups of a system into the ciphertext. Therefore, a subsequent partial decryption attempt by a DS only succeeds with user key-parts that satisfy the ciphertext's access policy. The RS is furthermore involved in the generation of the system parameters. It is also responsible for the key-part generation- and update-processes in partnership with the KA.

4.1.3. Storage Server

While not technically involved in any cryptographic operations in the ABE system, the Storage Server (StS) is mentioned in this section because data storage is an integral part of the proposed system. The prototype incorporates the StS inside the RS for simplicity. But there are no requirements that demand such a consolidation of these two servers, as long as the ciphertext upload is done via the RS.

4.1.4. Decryption Server

The Decryption Server (DS) is capable of partially decrypting ciphertexts obtained from a StS. For this operation, the DS requires the key-parts from KA and RS. This partial decryption performs all expensive bilinear pairing operations, while leaving a simple ElGamal [15] encrypted ciphertext for the Client (CL) to decrypt. Since it never gains access to a CL's key-part, the DS is not able to fully decrypt a ciphertext and thus can't acquire any information about the associated plaintext.

4.1.5. Data Owner

A Data Owner (DO) produces and encrypts information using a random secret and an access policy based on the ABE system's attributes. The random secret is split into several shares by using a LSSS, based on the access policy. These shares are applied to the initial ciphertext and can only be correctly recovered, if the attributes of a CL's private keys fulfill the access policy. This initial ciphertext is subsequently uploaded to the StS via the RS.

4.1.6. Client

A Client (CL) is an endpoint, that wants to access encrypted data generated in the ABE system. It is therefore necessary for the CL to participate in the update process of key-parts. If the key-parts associated with a CL fulfill the access policy of a ciphertext, it is possible to obtain the plaintext with help of the DS. For now, the CL does not provide input to the update protocol, as this would require a major redesign of the protocol and therefore be out of scope of this thesis. Input and modification steps during the process on the client side would be interesting to explore, as the key update would not exclusively rely on inputs from KA and RS.

4.2. ABE System

This section describes the ABE system generation and the protocols for ciphertext encryption/decryption. Furthermore we discuss generation and update of private key-parts. While based on the protocols of Lin, Hong, and Sun [38], there are some significant differences in the descriptions provided throughout this section. Lin et. al. based their protocols on bilinear pairings of Type I, as can be seen seen in several systems discussed in Chapter 3. For efficiency reasons, we chose a Type III pairing for implementing this system, specifically the *Ate Pairing over Barreto-Naehrig curves*. For more information we refer to Section 2.1. The pairing choice resulted in several changes throughout the descriptions and formulas the system is based on.

To better illustrate how we used the ECs of the chosen pairing in the actual implemented system, we will be using the notion of additive groups instead of multiplicative groups for the two paired groups $\mathbb{G}_1$ and $\mathbb{G}_2$. The target group remains a multiplicative group. Variable names in definitions are adopted from [38], for the most part.

4.2.1. System Definitions

Given two additive cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$ with generators P_1, P_2 respectively and a multiplicative group $\mathbb{G}_T$. Let the pairing map be:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T \tag{4.1}$$

The prime order of $\mathbb{G}_1$ is $p \in \mathbb{P}$. The hash functions of this system are defined as:

$$H_0 \colon \{0,1\}^* \to \mathbb{G}_1 \tag{4.2a}$$
$$H_1 \colon \mathbb{G}_1 \to {\mathbb{Z}_p}^* \tag{4.2b}$$
$$H_T \colon \mathbb{G}_T \to {\mathbb{Z}_p}^* \tag{4.2c}$$

The system defines attributes, that may be used to build ciphertext policies. This is denoted by the set $\hat{\mathcal{A}}$. All authorized system CLs are contained in the set $\mathcal{U}$. Each attribute is associated with an attribute group G, consisting of system CLs that hold this attribute. The set of all attribute groups is specified as $\mathcal{G}$.

$$\mathcal{U} = \{u_1, u_2, \ldots, u_n\} \tag{4.3a}$$
$$\hat{\mathcal{A}} = \{\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_m\} \tag{4.3b}$$
$$\mathcal{G} = \left\{G_i \mid \forall \hat{a}_i \in \hat{\mathcal{A}}, G_i \subseteq \mathcal{U}\right\} \tag{4.3c}$$

4.2.2. System Setup

The setup of an ABE system consists of three algorithms, that have to be executed to generate the system parameters *params* which are required to be publicly available to all system actors.

BaseSetup$(\lambda, \hat{\mathcal{A}})$ The initial setup algorithm for generating base system parameters takes a security parameter λ and the initial system attributes $\hat{\mathcal{A}}$. It then selects suitable groups for the system pairing based on λ and generates for every system attribute a corresponding element $A \in \mathbb{G}_1$. This is done by choosing a random element $a \in {\mathbb{Z}_p}^*$ and applying it to P_1:

$$A_i = a_i P_1 \tag{4.4a}$$
$$\mathcal{A} = \{A_i \mid \forall \hat{a}_i \in \hat{\mathcal{A}}\} \tag{4.4b}$$

Finally, the algorithm defines the generators of the pairing groups and outputs the base system parameters $params_{base}$:

$$params_{base} = \{P_1, P_2, \mathcal{A}, H_0, H_1, H_T\} \tag{4.5}$$

KeyAuthoritySetup$(params_{base})$ After the generation of $params_{base}$, the KA selects a random element $q \in {\mathbb{Z}_p}^*$ as its secret parameter and calculates a public parameter to complete its parameters $params_{KA}$:

$$params_{KA} = \{secret_{KA} = q, public_{KA} = qP_1\} \tag{4.6}$$

ReEncryptionSetup$(params_{base})$ After receiving $params_{base}$, the RS selects a random element $\alpha \in \mathbb{Z}_p{}^*$ as its secret parameter and calculates a public parameter to complete its parameters $params_{RE}$.:

$$params_{RE} = \left\{secret_{RE} = \alpha,\ public_{RE} = e\left(P_1, P_2\right)^{\alpha}\right\} \tag{4.7}$$

After the RS registers its public parameter with the KA, the final system parameters are created and distributed to all actors in the system:

$$params = \{params_{base},\ public_{KA},\ public_{RE}\} \tag{4.8}$$

4.2.3. Key Creation and Update

In this system, the private key of a client is split 3-ways, where KA, RS and the CL each possess a part of the key. The process for generating these key parts is based on two separate protocols. The first protocol only involves KA and RS, where both generate initial keys that are used in the second protocol to actually generate functioning private key-parts. The initial keys are kept for update purposes when CLs request new keys or the KA revokes attribute group memberships from CLs. In case a CL obtains a new attribute, both protocols have to be run again because the random ephemeral values used in the protocols are not persisted on the server side. Therefore, the key-parts can not be extended with matching attribute information. When revoking an attribute, the KA only needs to delete the attribute from its key-part and notify the RS to invalidate its key and rerun the second protocol.

4.2.3.1. Initial Key Protocol

Whenever it is required to build a CL's key from scratch, the KA will trigger the initial key protocol. The KA will generate a random value $\tau \in \mathbb{Z}_p{}^*$. This, along with its secret parameter q, will be its input to the protocol. The RS's input to the protocol will be its secret parameter α. The beginning of the protocol requires the computation of a value $x \in \mathbb{Z}_p{}^*$. To achieve this without revealing the servers' secret parameters, MPC comes into play.

$$x = (\alpha/q + \tau)/q = \frac{\alpha}{q^2} + \frac{\tau}{q} \tag{4.9}$$

After both servers obtained x, the RS chooses a random ephemeral value $\sigma \in \mathbb{Z}_p{}^*$ and calculates the first intermediate element V:

$$V = \frac{x}{\sigma} P_1 \tag{4.10}$$

V is sent to the KA, which in turn computes the intermediate element W:

$$W = q^2 V = q^2 \frac{x}{\sigma} P_1 \tag{4.11}$$

The KA sends W to the RS, which obtains its initial key IK_{RS} through following computation:

$$\begin{aligned} IK_{RS} &= \sigma W = \sigma q^2 \frac{x}{\sigma} P_1 = q^2 x P_1 \\ &= q^2 \left(\frac{\alpha}{q^2} + \frac{\tau}{q} \right) P_1 = (\alpha + q\tau) P_1 \end{aligned} \tag{4.12}$$

The intial key IK_{KA} is obtained by the KA based on the CL's set of attributes $\mathcal{S} \subseteq \hat{\mathcal{A}}$ and the ephemeral value τ:

$$IK_{KA} = \{\tau P_2, \forall \hat{a} \in \mathcal{S} \colon \tau A_{\hat{a}}\} \tag{4.13}$$

4.2.3.2. Update Key Protocol

When a CL requests a new key, for example when signing in to the system or after the servers invalidated their key-parts, the protocol for key updating is executed. Here, all 3 actors are involved in the MPC computation. The RS selects random values $r_1, \pi_1 \in \mathbb{Z}_p{}^*$ while the KA selects its random values $r_2, \pi_2 \in \mathbb{Z}_p{}^*$. The CL does not provide input to the protocol, but directly receives its private key-part from it.

$$y = (r_1 + r_2)\,\pi_1\pi_2 \tag{4.14}$$

While the servers receive the result $y \in {\mathbb{Z}_p}^*$, the CL obtains a different result PK_{CL} and uses that as its key-part for following decryption attempts.

$$PK_{CL} = \frac{(r_1 + r_2)}{\pi_1\pi_2} \tag{4.15}$$

After this step, the protocol continues with just the servers. The first key that will be computed belongs to the RS. The RS starts by selecting a random ephemeral $\xi \in {\mathbb{Z}_p}^*$ and applying it to its initial key IK_{RS}:

$$X_1 = \frac{y}{\xi} IK_{RS} = \frac{y}{\xi}\,(\alpha + q\tau)\,P_1 \tag{4.16}$$

The intermediate element X_1 is then sent to the KA which computes the intermediate value Y_1:

$$Y_1 = \frac{1}{\pi_2^2} X_1 \tag{4.17}$$

After receiving Y_1, the RS applies the following computation on the intermediate value to obtain its private key PK_{RS}:

$$\begin{aligned} PK_{RS} &= \frac{\xi}{\pi_1^2} Y_1 = \frac{\xi}{\pi_1^2}\frac{1}{\pi_2^2}\frac{(r_1 + r_2)\,\pi_1\pi_2}{\xi}\,(\alpha + q\tau)\,P_1 \\ &= \frac{r_1 + r_2}{\pi_1\pi_2}\,(\alpha + q\tau)\,P_1 \end{aligned} \tag{4.18}$$

The private key of the KA consists of multiple parts, just like its initial key. The KA starts by selecting a random ephemeral value $\varsigma \in {\mathbb{Z}_p}^*$ and computes the first intermediate elements based on its initial key IK_{KA}:

$$X_2 = \frac{y}{\varsigma}\tau P_2 \tag{4.19a}$$

$$\forall \hat{a} \in \mathcal{S}\colon X_{\hat{a}} = \frac{y}{\varsigma}\tau A_{\hat{a}} \tag{4.19b}$$

These elements are sent to the RS, which in turn computes the next intermediate elements:

$$Y_2 = \frac{1}{\pi_1^2}X_2 \tag{4.20a}$$

$$\forall \hat{a} \in \mathcal{S}\colon Y_{\hat{a}} = \frac{1}{\pi_1^2}X_{\hat{a}} \tag{4.20b}$$

After receiving these elements, the KA computes the elements that will form PK_{KA}:

$$\begin{aligned} Z &= \frac{\varsigma}{\pi_2^2}Y_2 = \frac{\varsigma}{\pi_2^2}\frac{1}{\pi_1^2}\frac{(r_1+r_2)\,\pi_1\pi_2}{\varsigma}\tau P_2 \\ &= \frac{(r_1+r_2)\,\tau}{\pi_1\pi_2}P_2 \end{aligned} \tag{4.21a}$$

$$\begin{aligned} \forall \hat{a} \in \mathcal{S}\colon Z_{\hat{a}} &= \frac{\varsigma}{\pi_2^2}Y_{\hat{a}} \\ &= \frac{\varsigma}{\pi_2^2}\frac{1}{\pi_1^2}\frac{(r_1+r_2)\,\pi_1\pi_2}{\varsigma}\tau A_{\hat{a}} \\ &= \frac{(r_1+r_2)\,\tau}{\pi_1\pi_2}A_{\hat{a}} \end{aligned} \tag{4.21b}$$

This leads to the KA's final key PK_{KA}:

$$\begin{aligned} PK_{KA} &= \{Z,\, \forall \hat{a} \in \mathcal{S}\colon Z_{\hat{a}}\} \\ &= \left\{\frac{(r_1+r_2)\,\tau}{\pi_1\pi_2}P_2,\, \forall \hat{a} \in \mathcal{S}\colon \frac{(r_1+r_2)\,\tau}{\pi_1\pi_2}A_{\hat{a}}\right\} \end{aligned} \tag{4.22}$$

4.2.4. Encryption

To encrypt data in the ABE system, a DO requires the system parameters *params*, an access structure $\mathbb{A}$ describing the access policy for the ciphertext, a secret $s \in \mathbb{Z}_p{}^*$ which will also be split into shares for encrypting the attributes and an element $\mathcal{M} \in \mathbb{G}_T$. $\mathcal{M}$ is normally defined as the plaintext in ABE systems. However, generic data cannot freely be translated back and forth into the group $\mathbb{G}_T$. Therefore, $\mathcal{M}$ will act as the input from which an AES key is derived. This key will then be used to produce a symmetrically encrypted payload PL that will be added to the ABE ciphertexts.

To create the payload key k_{PL}, a random element $\mathcal{M}$ and a random initialization vector IV are chosen as input for the hashing algorithm. The resulting hash will then be used as the key for a symmetric 256-bit AES encryption E of the plaintext data m.

$$k_{PL} = H_{SHA256}\left(IV,\, f_{bytes}\left(\mathcal{M}\right)\right) \tag{4.23a}$$

$$PL = \left\{IV,\, E_{k_{PL}}\left(m\right)\right\} \tag{4.23b}$$

An initial ciphertext CT_{init} consists of the payload PL, a share matrix M_{share} with labeled rows according to the LSSS from the access structure $\mathbb{A}$ and the element $\mathcal{M}$ encrypted using the public parameter $public_{RE}$. It furthermore includes the chosen ciphertext secret s, an element based on s and the encrypted attributes. These attributes are encrypted using the public parameter $public_{KA}$, s and the secret shares λ_t (entries from the share vector v_s, see Equation (A.3)) generated by the LSSS.

$$\begin{aligned} CT_{init} = \big\{ & PL,\, M_{share},\, C = \mathcal{M} \cdot e\left(P_1, P_2\right)^{\alpha s}, \\ & C_\perp = sP_2,\, \forall \hat{a}_t \in \mathbb{A} \colon C_t^* = \lambda_t q P_1 + \left(-s\right) A_t \big\} \end{aligned} \tag{4.24}$$

4.2.5. Re-Encryption

After DOs create their CT_{init}, they send it to the RS where it will be modified to build the full ciphertext CT. This new ciphertext incorporates CT_{init} and re-encrypts the attribute parts with so-called attribute group keys. To

recover these attribute group keys later on, the RS builds a ciphertext header. This allows the DS to solve equations based on the header's entries. On attempting to decrypt a ciphertext, the recovery succeeds for authorized CLs and fails for unauthorized CLs.

The RS generates two random values $\mu, \gamma \in \mathbb{Z}_p{}^*$ which are used to construct the header message Hdr. It also generates random attribute group keys $k_t \in \mathbb{Z}_p{}^*$ for each encrypted attribute ciphertext part C_t^*. This is done by selecting a random $z_t \in \mathbb{Z}_p{}^*$, calculating R_t and hashing it with H_T:

$$R_t = z_t P_1 \tag{4.25a}$$
$$k_t = H_1(R_t) \tag{4.25b}$$

To recover a key k_t, we have to create polynoms describing the authorized CLs of the corresponding attribute group G_t. In order to do so, we first need for all CLs u_k in the ciphertext's set of attribute groups $\mathcal{G}$ a corresponding element $x_k \in \mathbb{Z}_p{}^*$, based on a CL's unique identifying bitstring $ID_k \in \{0,1\}^*$.

$$\forall u_k \in \mathcal{G} \colon Q_k = H_0(ID_k) \tag{4.26a}$$
$$x_k = H_T(e(\mu Q_k, \gamma P_2)) \tag{4.26b}$$

Now the polynomial for each attribute group $G_t \in \mathcal{G}$ can be constructed through the roots of the polynomial, which are represented by the client elements $x_k \in \mathbb{Z}_p{}^*$ of all CLs $u_k \in G_t$. For the algorithm to calculate the polynomial, see Listing A.1. The mathematical definition of the polynomial function $f_t(x)$, with v being the number of CLs in G_t:

$$\begin{aligned} f_t(x) &= \prod_{i=1}^{v} (x - x_i) \\ &= \sum_{i=0}^{v} a_i x^i \end{aligned} \tag{4.27}$$

With the polynomial's factors a_i, we can construct an attribute group's polynomial tuple. This tuple will be blinded by a random factor $b \in {\mathbb{Z}_p}^*$ to build the header Hdr_t corresponding to the specific attribute ciphertext part C_t^*:

$$\{T_0, T_1, \ldots, T_v\} = \{a_0 P_1, a_1 P_1, \ldots, a_v P_1\} \tag{4.28a}$$

$$Hdr_t = \{R_t + bT_0, bT_1, \ldots, bT_v\} \tag{4.28b}$$

In the decryption process, the DS will be able to evaluate the polynomial using the requesting CL's element x_k, to retrieve the re-encryption element R_t for each attribute ciphertext part C_t^*. From this element, the DS will subsequently derive the attribute group key k_t, required for a successful decryption of C_t^*.

The entire header part will be constructed as follows:

$$Hdr = \{\mu P_2, \gamma, \forall G_t \in \mathcal{G} \colon Hdr_t\} \tag{4.29}$$

This header will be incorporated into the full ciphertext and the attribute ciphertext parts will be re-encrypted with their respective keys k_t:

$$\begin{aligned} CT = \{&Hdr, PL, M_{share}, \\ &C = \mathcal{M} \cdot e\left(P_1, P_2\right)^{\alpha s}, C_\perp = sP_2, \\ &\forall \hat{a}_t \in \mathbb{A} \colon C_t^* = k_t \left(\lambda_t q P_1 + (-s) A_t\right)\} \end{aligned} \tag{4.30}$$

4.2.6. Decryption

When a DS receives a decryption request from a CL, it immediately retrieves the key-parts PK_{KA} and PK_{RS} from the other servers. The CL's authorized attributes $\mathcal{S}$ can be extracted from PK_{KA}. The CL attaches CT to the decryption request. Using the header Hdr, the DS attempts to recover the attribute group keys $k_t \mid \hat{a}_t \in \mathcal{S}$. Since $\mathcal{S} \subseteq \mathbb{A}$ only has to fulfill the ciphertext policy, the DS tries to decrypt the relevant ciphertext parts $C_t^* \mid \hat{a}_t \in \mathcal{S}$ and ignores

all other ciphertext parts. The DS starts by computing the client element $x \in \mathbb{Z}_p{}^*$ based on its unique identity string ID using the information contained in Hdr:

$$Q = H_0(ID) \tag{4.31a}$$
$$x = H_T(e(\gamma Q, \mu P_2)) \tag{4.31b}$$

The element x of CL u is then applied to the polynomials in the relevant ciphertext parts $Hdr_t \mid \forall C_t^*$:

$$\begin{aligned}(R_t + bT_0) + \sum_{i=1}^{v} x^i bT_i &= R_t + ba_0 P_1 + \sum_{i=1}^{v} x^i ba_i P_1 \\ &= R_t + \left(b \underbrace{\sum_{i=0}^{v} x^i a_i}_{=0 \mid u \in G_t} \right) P_1 \\ &= R_t \mid u \in G_t\end{aligned} \tag{4.32}$$

The reencryption element R_t can only be recovered for an authorized CL $u \in G_t$, otherwise the computation results in a wrong element in $\mathbb{G}_1$. After successful recovery of R_t, we can calculate the attribute group key k_t (cf. Equation (4.25b)).

After obtaining the group keys k_t, the DS tries to recover the factors ω_t which are needed to restore the ciphertext secret s by solving a system of linear equations given the CL's authorized attributes $\mathcal{S}$. Recall the definition of the recover function in LSSS (see Equation (2.5)):

$$s = \sum_{t \mid \rho(t) \in \mathcal{S}} \omega_t \cdot \lambda_{\rho(t)} \tag{4.33}$$

For an example of this recovery, we refer to Appendix A.2. If $\mathcal{S}$ fulfills the policy described by $\mathbb{A}$, then the recovered factors will allow the DS to

perform its first decryption step using the information contained in key-part $PK_{KA} = \{Z, \forall \hat{a} \in \mathcal{S} \colon Z_{\hat{a}}\}$:

$$
\begin{aligned}
&\prod_{\rho(t)\in\mathcal{S}} \left(e\left(\frac{1}{k_t} C_t^*, Z\right) \cdot e\left(Z_{\rho(t)}, C_\perp\right)\right)^{\omega_t} \\
&= \prod_{\rho(t)\in\mathcal{S}} \left(e\left(\frac{1}{k_t} k_t \left(\lambda_t q P_1 + (-s) A_t\right), \frac{(r_1+r_2)\,\tau}{\pi_1\pi_2} P_2\right)\right. \\
&\quad \left. \cdot e\left(\frac{(r_1+r_2)\,\tau}{\pi_1\pi_2} A_t, s P_2\right)\right)^{\omega_t} \\
&= \prod_{\rho(t)\in\mathcal{S}} \left(e\left(\left(\lambda_t q - s a_t\right) P_1, \frac{(r_1+r_2)\,\tau}{\pi_1\pi_2} P_2\right) \cdot e\left(\frac{(r_1+r_2)\,\tau a_t}{\pi_1\pi_2} P_1, s P_2\right)\right)^{\omega_t} \\
&= \prod_{\rho(t)\in\mathcal{S}} e\left(P_1, P_2\right)^{\left(\frac{\lambda_t q \tau (r_1+r_2)}{\pi_1\pi_2} - \frac{s a_t \tau (r_1+r_2)}{\pi_1\pi_2} + \frac{s a_t \tau (r_1+r_2)}{\pi_1\pi_2}\right)^{\omega_t}} \\
&= \prod_{\rho(t)\in\mathcal{S}} e\left(P_1, P_2\right)^{\left(\frac{\lambda_t q \tau (r_1+r_2)}{\pi_1\pi_2}\right)^{\omega_t}} = \prod_{\rho(t)\in\mathcal{S}} e\left(P_1, P_2\right)^{\frac{\omega_t \lambda_t q \tau (r_1+r_2)}{\pi_1\pi_2}} \\
&= e\left(P_1, P_2\right)^{\frac{sq\tau(r_1+r_2)}{\pi_1\pi_2}} = F_0
\end{aligned}
\tag{4.34}
$$

The resulting element F_0 represents an intermediate partial decryption of $\mathcal{M}$ that is now independent of the attribute related ciphertext parts. To obtain the final partially decrypted element F_1, the DS has to do one final computation using the RS's key-part PK_{RS}:

$$\begin{aligned}
\frac{e\left(PK_{RS}, C_{\perp}\right)}{F_0} &= \frac{e\left(\frac{r_1+r_2}{\pi_1\pi_2}\left(\alpha+q\tau\right)P_1, sP_2\right)}{e\left(P_1,P_2\right)^{\frac{sq\tau\left(r_1+r_2\right)}{\pi_1\pi_2}}} \\
&= \frac{e\left(P_1,P_2\right)^{\frac{s\alpha\left(r_1+r_2\right)}{\pi_1\pi_2}+\frac{sq\tau\left(r_1+r_2\right)}{\pi_1\pi_2}}}{e\left(P_1,P_2\right)^{\frac{sq\tau\left(r_1+r_2\right)}{\pi_1\pi_2}}} \\
&= e\left(P_1,P_2\right)^{\frac{s\alpha\left(r_1+r_2\right)}{\pi_1\pi_2}+\frac{sq\tau\left(r_1+r_2\right)}{\pi_1\pi_2}-\frac{sq\tau\left(r_1+r_2\right)}{\pi_1\pi_2}} \\
&= e\left(P_1,P_2\right)^{\frac{s\alpha\left(r_1+r_2\right)}{\pi_1\pi_2}} = F_1
\end{aligned} \tag{4.35}$$

The final resulting element F_1 is then incorporated into the partially decrypted ciphertext CT_{PD} which is returned to the CL that made the decryption request:

$$CT_{PD} = \{PL,\, F_1,\, C\} \tag{4.36}$$

After receiving CT_{PD}, the CL performs the last computation to obtain the original element $\mathcal{M}$, which is needed to decrypt the payload PL:

$$\begin{aligned}
\frac{C}{F_1^{\frac{1}{PK_{CL}}}} &= \frac{C}{F_1^{\frac{\pi_1\pi_2}{r_1+r_2}}} = \frac{\mathcal{M}\cdot e\left(P_1,P_2\right)^{\alpha s}}{\left(e\left(P_1,P_2\right)^{\frac{s\alpha\left(r_1+r_2\right)}{\pi_1\pi_2}}\right)^{\frac{\pi_1\pi_2}{r_1+r_2}}} \\
&= \frac{\mathcal{M}\cdot e\left(P_1,P_2\right)^{\alpha s}}{e\left(P_1,P_2\right)^{\frac{s\alpha\left(r_1+r_2\right)}{\pi_1\pi_2}\frac{\pi_1\pi_2}{r_1+r_2}}} = \frac{\mathcal{M}\cdot e\left(P_1,P_2\right)^{\alpha s}}{e\left(P_1,P_2\right)^{s\alpha}} \\
&= \mathcal{M}\cdot e\left(P_1,P_2\right)^{\alpha s}\cdot e\left(P_1,P_2\right)^{-\alpha s} \\
&= \mathcal{M}
\end{aligned} \tag{4.37}$$

At last, the CL can compute the payload key k_{PL} using $\mathcal{M}$ and IV (see Equation (4.23a)). With this key, the CL can decrypt the original AES encrypted plaintext m.

4.2.7. Attribute Management

Whenever a CL obtains or has an attribute $\hat{a}_i$ revoked, the corresponding attribute group G_i has to be modified to reflect the current set of authorized CLs. This updated group will be denoted G_i'. Since attribute group keys of ciphertexts depend on the clients $u \in G_i$, ciphertexts have to be updated accordingly. Due to the structure of a final ciphertext CT and its header Hdr, the update process only has to alter the parts affected by said attribute group. A new attribute group key k_i' has to be applied to the corresponding ciphertext part C_i^* and a new attribute header Hdr_i is created based on the current clients in G_i'.

We will now present the methods required to handle changes in users' attributes.

4.2.7.1. Key Update

The servers' key-parts have to be updated after client attributes are revoked or issued. In case of revocation, the KA removes the revoked attribute $\hat{a}_r$ from the CL's authorized attribute set $\mathcal{S}$:

$$\mathcal{S}' = \mathcal{S} \setminus \{\hat{a}_r\} \tag{4.38}$$

It then drops the attribute $\hat{a}_r$ from its initial key IK_{KA}:

$$IK'_{KA} = \left\{\tau P_2, \forall \hat{a} \in \mathcal{S}' \colon \tau A_{\hat{a}}\right\} \tag{4.39}$$

The KA subsequently instructs the RS to rerun protocol Section 4.2.3.2, where both servers provide new input values, namely $r_1', \pi_1', r_2', \pi_2' \in \mathbb{Z}_p{}^*$.

This leads to new key-parts for all involved actors:

$$PK'_{CL} = \frac{(r'_1 + r'_2)}{\pi'_1 \pi'_2} \tag{4.40a}$$

$$PK'_{RS} = \frac{r'_1 + r'_2}{\pi'_1 \pi'_2} (\alpha + q\tau) P_1 \tag{4.40b}$$

$$PK'_{KA} = \left\{ \frac{(r'_1 + r'_2)\,\tau}{\pi'_1 \pi'_2} P_2,\ \forall \hat{a} \in \mathcal{S}' :\ \frac{(r'_1 + r'_2)\,\tau}{\pi'_1 \pi'_2} A_{\hat{a}} \right\} \tag{4.40c}$$

In case a CL obtains an attribute $\hat{a}_g$, the keys have to be created anew, as the value of τ is not retained after the protocols are finished. Therefore, the initial key protocol Section 4.2.3.1 has to be executed first. Initially, the KA updates the user's attribute set:

$$\mathcal{S}' = \mathcal{S} \cup \{\hat{a}_g\} \tag{4.41}$$

Subsequently, the KA chooses a new value $\tau' \in \mathbb{Z}_p{}^*$ and triggers protocol Section 4.2.3.1, which leaves both servers with new initial keys:

$$IK'_{RS} = (\alpha + q\tau')\, P_1 \tag{4.42a}$$

$$IK'_{KA} = \{\tau' P_2,\ \forall \hat{a} \in \mathcal{S}' :\ \tau' A_{\hat{a}}\} \tag{4.42b}$$

Executing protocol Section 4.2.3.2 results in the following updated key-parts for the participating actors:

$$PK'_{CL} = \frac{(r'_1 + r'_2)}{\pi'_1 \pi'_2} \tag{4.43a}$$

$$PK'_{RS} = \frac{r'_1 + r'_2}{\pi'_1 \pi'_2} (\alpha + q\tau') P_1 \tag{4.43b}$$

$$PK'_{KA} = \left\{ \frac{(r'_1 + r'_2)\,\tau'}{\pi'_1 \pi'_2} P_2,\ \forall \hat{a} \in \mathcal{S}' :\ \frac{(r'_1 + r'_2)\,\tau'}{\pi'_1 \pi'_2} A_{\hat{a}} \right\} \tag{4.43c}$$

4.2.7.2. Ciphertext Update

When attribute groups change $\left(G_l \rightarrow G_l'\right)$, ciphertexts containing said groups have to be updated in response. Group keys k_l should only be recoverable during requests of authorized CLs $u_x \in G_l$. Therefore, new group keys k_l' are chosen to re-encrypt the corresponding ciphertext parts. These keys are protected by generating new ciphertext headers Hdr_l'. In order to do so, the RS has to retrieve the original ciphertext part C_l^* from CT_{init}, re-encrypt it with k_l' and replace it in CT.

First, a new element R_l and a subsequent re-encryption key have to be calculated:

$$R_l' = z_l' P_1 \tag{4.44a}$$
$$k_l' = H_1\left(R_l'\right) \tag{4.44b}$$

Then, a new header for the changed group G_l' has to be built, by creating a polynomial based on the group's updated set of contained client elements x_k:

$$\begin{aligned} f_l(x) &= \prod_{i=1}^{v'} (x - x_i) \\ &= \sum_{i=0}^{v'} a_i' x^i \end{aligned} \tag{4.45}$$

With a new random blinding factor b', the header Hdr_l' is constructed:

$$\left\{T_0', T_1', \ldots, T_{v'}'\right\} = \left\{a_0' P_1, a_1' P_1, \ldots, a_{v'}' P_1\right\} \tag{4.46a}$$
$$Hdr_l' = \left\{R_l' + b' T_0', b' T_1', \ldots, b' T_{v'}'\right\} \tag{4.46b}$$

This leads to a new header Hdr':

$$Hdr' = \left\{ \mu P_2,\, \gamma,\, Hdr'_l,\, \forall G_t \in \mathcal{G} \setminus \{\hat{a}_l\} : Hdr_t \right\} \tag{4.47}$$

This header is then incorporated into the updated ciphertext CT' along the newly re-encrypted ciphertext part $C_l^{*\prime}$:

$$\begin{aligned} CT' = \big\{ & Hdr',\, PL,\, M_{share}, \\ & C = \mathcal{M} \cdot e\left(P_1, P_2\right)^{\alpha s},\, C_\perp = sP_2, \\ & C_l^{*\prime} = k_l{}' \left(\lambda_l q P_1 + (-s)\, A_l\right), \\ & \forall \hat{a}_t \in \mathbb{A} \setminus \{\hat{a}_l\} : C_t^* = k_t \left(\lambda_t q P_1 + (-s)\, A_t\right) \big\} \end{aligned} \tag{4.48}$$

This chapter gave the mathematical definitions for the ABE system presented in this thesis. The following chapter will go into details on how this system was implemented.

5. Prototype

While the previous chapter detailed the mathematical operations required for the presented ABE system, it does not describe how such a system would be realized in practice. The prototype provides this functionality by implementing all relevant actors.

This chapter will get into technical details of the prototype systems. Used environments, tools and libraries.

Due to the chosen architecture, the prototype consists of several components. It's organized with *Gradle Build Tool* [25]. Each component is a Gradle project in itself, with some interdependencies between them. Communication throughout the system happens among various lines as Figure 5.1 shows. All actors incorporate those ABE libraries necessary for their operations. The server components store data in their respective *MongoDB* databases. Communication with servers is done via their exposed Representational State Transfer (REST) Application Programming Interfaces (APIs). To securely generate and update keys, separate sockets have to be used for handling MPC with *FRESCO*. The server components are managed via a web interface, while the producer and client implementations are Command Line Interface (CLI) applications.

5.1. External Dependencies

The prototype components depend on certain external tools that were necessary to implement the desired ABE functionality. These dependencies are described in the following sections.

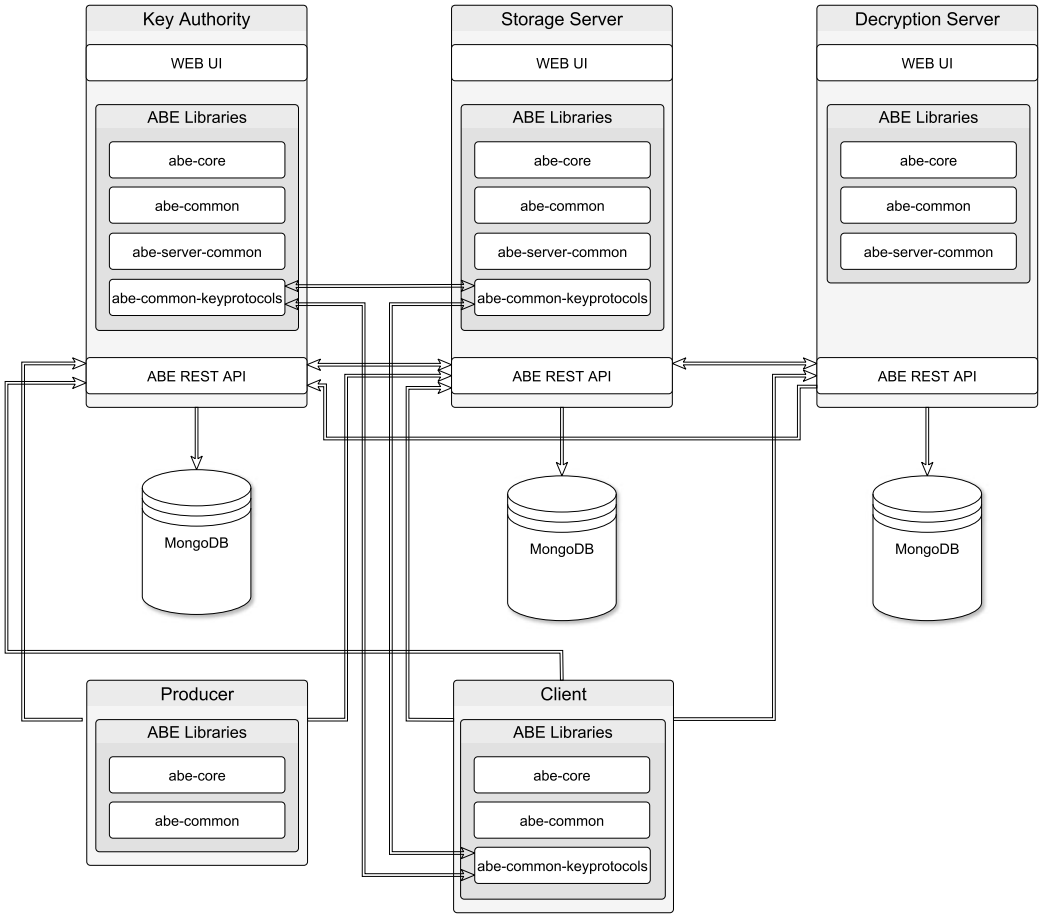

Figure 5.1.: Overview of the entire prototype

5.1.1. Cryptographic Libraries

The implementation of pairings was provided by the *ECCelerate*^{TM} [14] library from the Institute of Applied Information Processing and Communications (IAIK) of the Graz University of Technology (TU Graz). The library provides an implementation for the Ate pairing over BN curves in the asymmetric Type II and Type III setting. As our architecture does not require an isomorphism, only pairings of the Type III variant were used.

The hashing function H_0 is supplied by the library *ECCelerate*^{TM} [14] with description of the implementation in the master's thesis of Ramacher [46]. The underlying algorithm to securely and efficiently hash onto Barreto-Naehrig curves is based upon work from Fouque and Tibouchi [18]. The hash functions H_1 and H_T were chosen to be a 256 bit Secure Hash Algorithm (SHA) digest of the unique byte representation of elements from the groups $\mathbb{G}_1$ and $\mathbb{G}_T$ reduced by p, for simplicity's sake. This means, that the hashes are tied to the implementation's use of the *ECCelerate*^{TM} framework and its current choice of internal representation of elements.

$$x_1 \in \mathbb{G}_1 \colon H_1(x_1) \equiv H_{SHA256}\left(f_{bytes}(x_1)\right) \pmod{p} \tag{5.1a}$$

$$x_T \in \mathbb{G}_T \colon H_T(x_T) \equiv H_{SHA256}\left(f_{bytes}(x_T)\right) \pmod{p} \tag{5.1b}$$

AES and SHA-256 implementations were also provided by IAIK through their JCE library [30]. These cryptographic tools were required for the generation of ciphertext payloads and hashing of group elements into the $\mathbb{Z}_p{}^*$ domain.

5.1.2. Multi-Party Computation

The prototype realizes the generation and update of private keys via MPC provided by the *FRESCO* [19] library. It implements various MPC protocols in the boolean and arithmetic setting. Since the private keys in the prototype are dependent on computations in $\mathbb{Z}_p{}^*$, the SPDZ [11] protocol was employed in the prototype.

5.1.3. Web Frameworks

Spring Boot [56] is an extension to Spring, a popular framework for building web applications in Java. It allows for automated configuration of Spring components and easy deployment through built-in containers.

MongoDB [43] is a so called NoSQL database system. Contrary to traditional relational database systems, *MongoDB* stores data in documents with a flexible JavaScript Object Notation (JSON)-like format.

Thymeleaf [57] is a template engine for rendering web pages server-side. It allows for front-end development with pure Hypertext Markup Language (HTML) templates and integrates well with the Spring MVC framework.

5.2. Libraries

This section lists all libraries that we developed to realize the thesis' prototype. They represent the core functionality of the ABE system and are incorporated in the actors' implementations.

5.2.1. abe-core

The project `abe-core` stands at the center of the prototype and implements the cryptographic high-level operations needed for the chosen ABE system. It furthermore provides the required classes like parameters, keys and access structures. LSSS matrix generation is implemented according to the method presented by Lewko and Waters [37]. Solving systems of linear equations needed for LSSS factor recovery was achieved with the use of Gauss-Jordan elimination (see Appendix A.2).

System Initialization To initialize an ABE system in the prototype, an object containing the global system parameters is required. The system settings are created by the KA and RS (see Section 4.2.2). Listing 5.1 shows a minimal example of initializing an ABE system with our prototype. The variable `security` refers to the bit size of the ECs that will be used to create a pairing. `HashType` refers to the method of hashing elements int $\mathbb{Z}_p{}^*$ (see Equation (5.1)). In combination with the set of attributes in the ABE system, we can create the `BaseSystemParameters`, representing an implementation of Equation (4.5). The following lines of code show how to instatiate KA and RS parameters according to Equation (4.6) and Equation (4.7). Subsequently, we get an object that holds all public system parameters described in Equation (4.8). The last line of this listing represents an object providing all methods that the system actors require.

Listing 5.1: Initialization of an ABE system

```
1  // defining the basic system parameters
2  int security = 160;
3  Set<String> systemAttributes = Stream.of("management", "quality", "
       purchasing", "operations", "factory").collect(Collectors.toSet());
4  HashTypes hashType = HashTypes.SIMPLE_SHA256;
5
6  // Basic system initialization
7  BaseSystemParameters baseSystemParameters = new BaseSystemParameters(
       security, systemAttributes, hashType);
8
9  // Key Authority parameter initialization
10 KeyAuthorityParameters keyAuthorityParameters = new
       KeyAuthorityParameters(baseSystemParameters.getSystemPairing().
       getGroup1());
11
12 // Reencryption parameter initialization
13 ReencryptionParameters reencryptionParameters = new
       ReencryptionParameters(baseSystemParameters.getSystemPairing());
14
15 // Creating the system parameters and ABE object
16 SystemParameters systemParameters = new SystemParameters(
       baseSystemParameters, keyAuthorityParameters.getPublicParameter(),
```

```
    reencryptionParameters.getPublicParameter());
17 AttributeBasedEncryption abe = new AttributeBasedEncryption(
    systemParameters);
```

Encryption A DO can encrypt data using a chosen policy. Here, the only prerequisite is an initialized ABE object. The policy has to be provided in the form of an object describing an access structure. The access structure is a tree-like object that has to be initialized with a string in postfix notation. Listing 5.2 shows the creation of an access policy from a policy string and the subsequent encryption of data according to Section 4.2.4.

Listing 5.2: Encryption of a ciphertext

```
1 String plaintext = "This is a standard test plaintext.";
2 String policy = "( management, ( ( ( operations, quality, AND), (
    purchasing, factory, AND), OR), ( (operations, quality, OR), (
    purchasing, factory, OR), AND ), OR ), AND )";
3 AccessStructure accessStructure = AccessStructure.
    createPolicyFromString(policy);
4
5 BasicCiphertext basicCiphertext = abe.encrypt(plaintext,
    accessStructure, StandardCharsets.UTF_8);
6
```

The encryption method takes in a parameter indicating the encoding to be used for the conversion of the plaintext string into a byte array. A method accepting the plaintext in the form of a byte-array is also available.

Re-Encryption The basic ciphertext from a DO can not be decrypted directly. It first has to be re-encrypted by the RS (see Section 4.2.5). For that, the attribute groups (see Equation (4.3)) have to be passed to the method by the RS, as is shown in Listing 5.3. The resulting ciphertext can then be stored and retrieved for decryption.

Listing 5.3: Re-Encryption of a ciphertext

```
1 Ciphertext ciphertext = abe.reencryptBasicCiphertext(basicCiphertext,
      attributeGroups);
```

Partial Decryption After a CL initiates a decryption request, the DS has to collect the corresponding private key-parts from KA and RS. Listing 5.4 shows how the DS uses the private key-parts as well as the user ID to perform a partial decryption of the ciphertext (see Section 4.2.6), which is then returned to the CL.

Listing 5.4: Partial decryption of a ciphertext

```
1 PartiallyDecryptedCiphertext partiallyDecryptedCiphertext = abe.
      decryptPartially(ciphertext, cpk1, cpk2, userID);
```

Decryption After obtaining a partially decrypted ciphertext from the DS, Listing 5.5 shows how a client makes the final decryption step (see Equation (4.37)) using its own private key-part.

Listing 5.5: Decryption of a ciphertext

```
1 String recoveredPlaintext = abe.decrypt(partiallyDecryptedCiphertext,
      StandardCharsets.UTF_8, cpk3);
```

5.2.2. abe-common

The `abe-common` library provides classes that are shared among all other prototype components. This includes general configuration and constants for the entire system. The main functionality it provides to other components revolves around serialization of ABE objects in a web-based environment.

The prototype implements object representations - so called models - in JSON format. All members of an object are serialized into JSON elements using the *Jackson* [33] mapper. One special case is the handling of binary data like byte arrays, which are converted into a *Base64* [4] format.

Depending on the members of ABE objects, the conversion of objects to models requires information about the system pairing. This requirement is based in how *ECCelerate*^{TM} represents points on ECs and field elements. These objects can be encoded into byte arrays. But they have to be decoded by the same ECs or fields of the system pairing. Therefore, the system pairing has to be present, before any affected ABE object can be recovered from its JSON form. Listing 5.6 demonstrates, that the `SystemPairing` object itself can be recovered directly. However, instances of certain private key objects need the system pairing to successfully deserialize its corresponding JSON model.

Listing 5.6: Conversions of JSON models to core classes

```
1 // a conversion that does not depend on already set-up system
     parameters
2 SystemParameters systemParameters = systemParameterModel.toCoreObject()
3
4 // a conversion that depends on the system pairing (or groups thereof),
      due to - for example - ECPoint members
5 PrivateKey privateKey = privateKeyModel.toCoreObject(systemParameters.
     getBaseSystemParameters().getSystemPairing());
```

All models can furthermore be equipped with some form of integrity data, with currently having salted SHA-256 hashes available in the implementation (see Listing B.2).

5.2.3. abe-common-keyprotocols

The `abe-common-keyprotocols` library contains the implementation for the creation and update of keys in the ABE prototype.

MPC Our implementation relies on the *FRESCO* [19] library to mutually agree on initial values required to create and update private key-parts, as was shown in Equation (4.9) and Equation (4.14). To arithmetically calculate these initial values during key creation and update, the SPDZ [11] protocol suite was chosen. A computation with *FRESCO* is called an application. The MPC players have to create objects from the same application class, instantiated with their individual inputs. The class itself describes the computation process and defines which intermediate or end results are opened to which player. If a player were to manipulate the computation or communication channel, *FRESCO* is able to detect this behavior and abort the computation. The same would hold, if someone were to alter their application object to try to open a value that is not intended for them. *FRESCO* works with Java sockets to communicate between players. The `abe-common-keyprotocols` library provides a class that manages all things related to the setup of communication between players, the so called `MPCPlayerSocketHandler`.

Key Protocols Key creation and update are built upon the computation of initial values, as was described in Section 4.2.3. The protocols incorporate the previously mentioned MPC applications and use these values to calculate the resulting keys. Since *FRESCO* requires complete control over the sockets used during MPC operations, the protocol classes need to establish a second connection to continue the remaining key manipulation steps. These additional connections are always limited to the KA and RS, since CLs only require results from the MPC portion of a key update.

5.2.4. abe-server-common

The `abe-server-common` library contains classes that are shared among all server components. Since servers have to establish trusted connections to one another, this library provides implementations for setting up said server relations for ABE systems. They also provide classes that handle the storage of these relations in *MongoDB* databases.

5.3. Servers

The servers are self-contained *Spring Boot* applications. They expose their ABE functionality through REST APIs (see Appendix B) and provide Graphical User Interfaces (GUIs) for management tasks. The involved servers can be defined on a system-by-system basis. This means, that a DS can provide its decryption service for ABE systems managed by different KAs. The same holds for the other servers as well. It should be mentioned, that the prototype combines the functionality of RS and StS into one server project.

5.3.1. abe-server-keyauthority

The `abe-server-keyauthority` component manages users, ABE systems and defines the associated StS and DS. Most of its functionality is exposed on REST endpoints. All systems are identified by their Universally Unique Identifier (UUID), which has to be passed as a parameter on each request to the KA's API. A list of exposed endpoints can be found in Appendix B.1. This component persists data in *MongoDB*. Examples of stored data are ABE systems, users, private keys and the corresponding initial private keys as well as a list of known trusted servers.

5.3.2. abe-server-storage

The `abe-server-storage` prototype component combines both RS and StS, as the re-encryption of ciphertexts and it's storage are so closely related. Similar to the KA, it provides a REST API (see Appendix B.2) for handling ciphertexts in the system. When a ciphertext is uploaded, the component performs the necessary re-encryption steps (see Section 4.2.5) before storing it permanently. Persistent data is handled via *MongoDB*, similar to the KA component. The major difference is the additional storage of ciphertexts that are linked to storage paths. The RS queries the KA for information on users and groups or gets notifications via it's dedicated REST endpoints.

5.3.3. abe-server-decryption

The DS component exposes one REST endpoint (see Appendix B.3), where CLs can submit ciphertexts for partial decryption. The DS then queries the KA and RS servers components for their private key-parts corresponding to the CL to perform the partial decryption of the submitted ciphertext. The component only permanently stores which other severs are associated to each ABE system.

5.4. Endpoints

The endpoint components implement basic functionality of how the ABE system would be used in practice.

5.4.1. abe-example-information

As the ABE system is agnostic to the actual data that will be encrypted and transferred, the `abe-example-information` component implements a common JSON data structure that is used by the DO and CL components.

5.4.2. abe-example-producer

`abe-example-producer` represents what an example DO would have to implement for interacting with a ABE system. The component regularly generates JSON data in the form of the common data structure from the previous section. These data structures are then encrypted according to the system's public parameters and submitted to the RS component under a specified data path. The required system parameters are retrieved via REST calls to the KA's API.

5.4.3. abe-example-client-java

The `abe-example-client-java` component is configured to retrieve data under given data paths from the RS in regular intervalls. On startup, it requests a new private key from the KA which will then trigger a MPC computation. When new data under a specific storage path can be retrieved, this data is submitted to the DS. The result is then used to decipher the original plaintext using the CL's private key according to Equation (4.37).

This concludes the prototype chapter. It gave an overview of all the implemented components and showed code examples to demonstrate how to achieve the functionality described in Chapter 4. The following chapter will present performance data generated by the prototype components and give an idea regarding its applicability in an IoT environment.

6. Evaluation

To determine if our system can be adopted in an IIoT environment, we performed an evaluation of key aspects of our prototype. In Section 6.1 we take a look at the inherent overhead our ABE architecture adds to otherwise symmetrically encrypted data. A thorough computational performance evaluation of the core cryptographic operations on hardware representing IIoT devices was conducted by Ziegler, Sabongui, and Palfinger [65]. We will point to these findings for high level ABE operations throughout Section 6.2. Furthermore, we examine execution times of several important building blocks of these operations.

6.1. Ciphertext Overhead

The output of CP-ABE can be seen as a symmetrically encrypted ciphertext, that has been enriched with information about its respective decryption policy. First, recall the definition of an initial ciphertext CT_{init} from Equation (4.24). As the size of the payload PL only depends on a 256-bit encryption with AES, we will not further consider it in the subsequent evaluation. The reason behind it being, that the ABE related portions of the full ciphertext are completely independent of the actual payload data, since its purpose is to protect the AES key of the payload.

We define the notation for the size of basic elements in bytes that constitute our ciphertext as $|\mathbb{Z}_p{}^*|$, $|\mathbb{G}_1|$, $|\mathbb{G}_2|$ and $|\mathbb{G}_T|$. The size of elements in our prototype is tied to the chosen security parameter λ. This parameter indicates the security strength of our system in bits. As p is the order of the resulting groups $\mathbb{G}_1$ and $\mathbb{G}_2$, we see that $|\mathbb{Z}_p{}^*| = |p| = \frac{\lambda}{8}$. The size of the

λ	$\lvert\mathbb{G}_1\rvert$	$\lvert\mathbb{G}_2\rvert$	$\lvert\mathbb{G}_T\rvert$
160	168	328	1 920
192	200	392	2 304
224	232	456	2 688
256	264	520	3 072
384	392	776	4 608
512	520	1 032	6 144

Table 6.1.: Bitsize of elements in *ECCelerate*^{TM} according to security parameter λ.

other elements is defined by their internal representation in *ECCelerate*^{TM}. Table 6.1 gives an overview of the sizes in bits at different security levels.

If we recall the definition of the initial ciphertext CT_{init} from Equation (4.24), we see that there are two parts that grow in size based on the number of attributes in an access structure $\mathbb{A}$. We will denote this quantity as $|\mathbb{A}|$. To generate the matrix, we use the method from Lewko and Waters [37]. For an example of this matrix generation, see Appendix A.2. Here, a sharing matrix M_{share} will always have $|\mathbb{A}|$ rows, while the columns depend on the concrete access policy. Each time an AND gate is encountered, the matrix will gain an additional column. An upper bound would be $|\mathbb{A}|$ in case the policy consists only of AND gates. However, the resulting matrices are always sparse, as a lot of the access tree labeling involves padding of vectors with zeroes. In general, we can say that there are two entries per row on average. Since the size of a zero value is always 4 bytes (due to it being a 1-element internal integer array of a `BigInteger`), we will approximate the size of the matrix with $2 \cdot |\mathbb{A}| \cdot |\mathbb{Z}_p{}^*| + |\mathbb{A}|^2 \cdot 4$. One last thing to consider is the size of the attribute labeling ρ for M_{share} and the list of C_t^*. As the implementation allows for any character string to be an attribute identifier, this can become a significant factor with increasing complexity of the access policy. Equation (6.1) shows the size we can expect from CT_{init} (excluding the payload PL):

$$|CT_{init}| = 2 \cdot |\mathbb{A}| \cdot |\mathbb{Z}_p{}^*| + |\mathbb{A}|^2 \cdot 4 + |\mathbb{G}_T| + |\mathbb{G}_2| + |\mathbb{A}| \cdot |\mathbb{G}_1| + 2 \cdot |\rho| \quad (6.1)$$

We can assume, that the size of the basic ciphertext will be dominated

by its access structure $\mathbb{A}$, and increasing complexity thereof. Even though zero values in the resulting matrix M_{share} need significantly less space than other elements, the (worst case) square nature of M_{share} can add respectable overhead with an increasing amount of attributes in $\mathbb{A}$.

A re-encrypted ciphertext adds Hdr to generate a complete ciphertext, see Equation (4.29). Since each Hdr_t protecting an attribute re-encryption key depends on the concrete number of CLs in an attribute group G_t, it is hard to give a general estimation. We will define the the total number of elements in $\forall Hdr_t \in Hdr$ as c. So our size approximation for a full ciphertext CT becomes:

$$\begin{aligned} |Hdr| &= |\mathbb{G}_2| + |\mathbb{Z}_p{}^*| + c \cdot |\mathbb{G}_1| + |\rho| \\ |CT| &= |CT_{init}| + |Hdr| \end{aligned} \tag{6.2}$$

While this gives us an approximation to the ciphertext overhead with the library `abe-core`, the transfer in our prototype happens in a JSON format. The size of this ciphertext cannot be completely defined in a general form, as the data conversion to JSON adds additional information, for example variable names, for better human readability. On the other hand, it brings certain values into a more compact form, like zeroes in our matrix which only require one byte to represent the character '0' compared to four bytes in a `BigInteger`.

Therefore, we will now turn to the outcome of the tests presented by Ziegler, Sabongui, and Palfinger [65]. They give concrete results for sizes of ciphertexts in their form targeted for JSON serialization, provided by the model classes of `abe-common`. The growth of ciphertext sizes can be seen in Figure 6.1. They evaluated for common EC security parameters (λ in our system respectively) and access policies with up to 100 attributes. The amount of CLs in each attribute group G_t was uniformly set to 5. Their results validate the previous assertion that ciphertext size is dominated by its access structure complexity, with the secondary factor being the increased element sizes with growing security parameter λ.

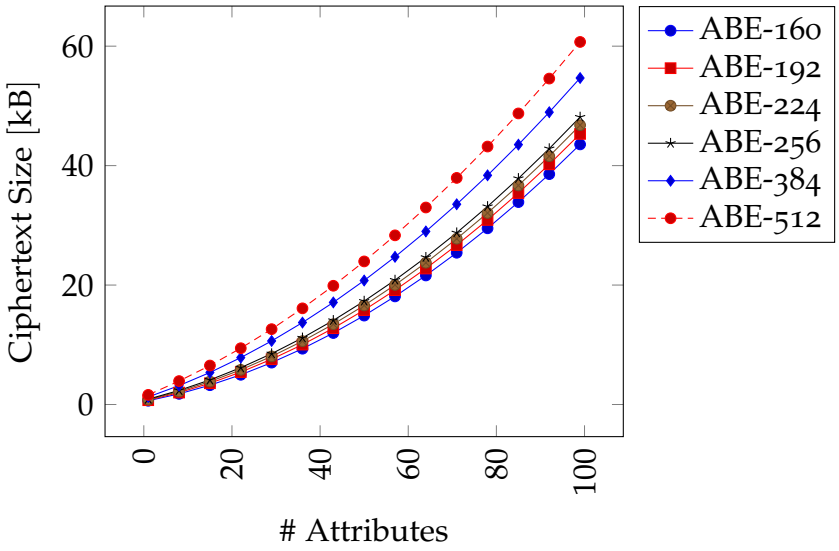

Figure 6.1.: Ciphertext size of ABE with prime order 160-bit to 512-bit. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz. From "Fine-Grained Access Control in Industrial Internet of Things: Evaluating Outsourced Attribute-Based Encryption" [65]

6.2. Computational Performance

For their evaluation of the high-level ABE operations, [65] used an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz to represent cloud services and a Raspberry Pi 3 Model B+ Central Processing Unit (CPU) to simulate a resource constrained IIoT device. We provide additional measurements of certain building blocks computed on the same CPU. If not stated otherwise, data points in figures represent the median execution time over 100 iterations.

The system architecture was chosen to accommodate for the range of computational power levels in the IIoT setting. Our goal was to unburden decrypting endpoints from performing bilinear pairings altogether. This results in a final decryption step with constant time duration, as this step always only requires one exponentiation and division each in $\mathbb{G}_T$. The evaluation in [65] showed, that a resource limited device can perform this computation at a security level of 256 bits in around 150ms. A cloud component with a potent processor requires only 5ms for the same operation

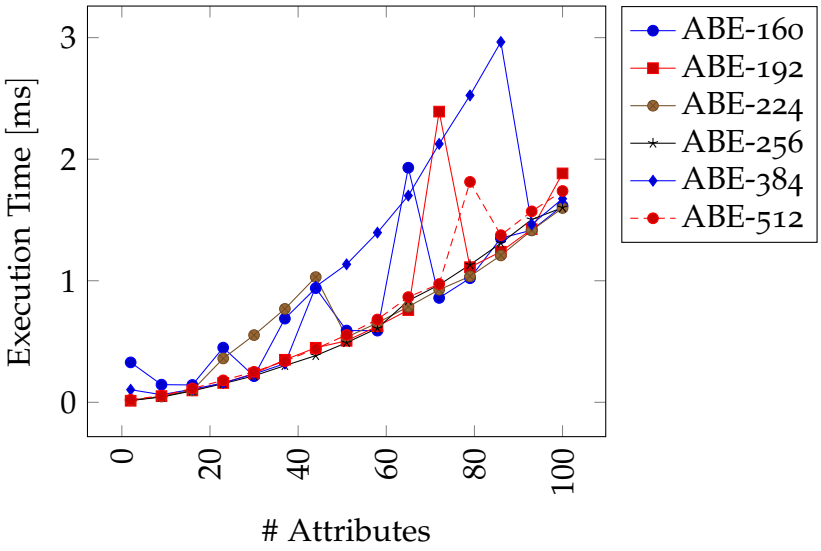

Figure 6.2.: Secret sharing setup duration of ABE with prime order 160-bit to 512-bit. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz

and 30ms at a 512 bit security level. As this operation is independent of access structure complexity, we turn to the interesting computations that are encryption, re-encryption and partial decryption.

Basic Encryption Recall Equation (4.24) and examine the needed operations to create a basic ciphertext. First, we have to create M_{share} and the vector v_s containing the secret shares, based on the complexity of the access structure $\mathbb{A}$. However, Figure 6.2 shows that our CPU, the execution time always stays below 3 ms. Due to the low execution times, we can see a few outliers that we assume to be the result of concurrently occurring computations unrelated to our test parameters.

We see, that the setup of our secret sharing components is negligible compared to the calculations in $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$. Here, we have to perform one exponentiation and multiplication in $\mathbb{G}_T$, one multiplication in $\mathbb{G}_2$, as well as two multiplications with one addition in $\mathbb{G}_1$ per attribute in $\mathbb{A}$. The evaluation of the initial encryption operation in [65] was conducted on both a powerful cloud service and a resource constrained IoT device. Due to the

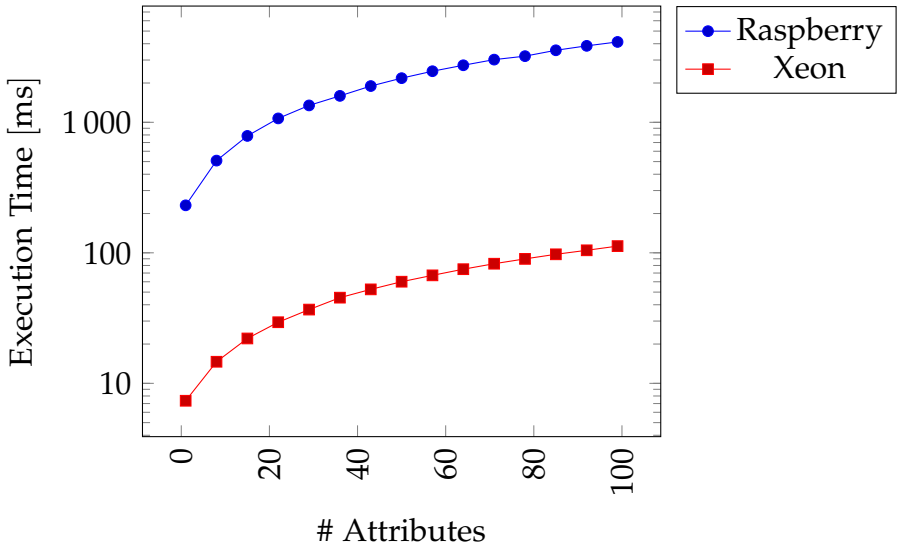

Figure 6.3.: Execution times of ABE encryption with prime order 256-bit on a logarithmic scale. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz and Raspberry Pi 3 Model B+ respectively. From "Fine-Grained Access Control in Industrial Internet of Things: Evaluating Outsourced Attribute-Based Encryption" [65]

IoT device's computational restrictions, they targeted a security parameter $\lambda = 256$, see Figure 6.3 for a comparison of execution times at this security level.

We see that the IoT device is quickly reaching its limit at this security level, with execution times surpassing one second at around 20 attributes and taking up to 4 seconds at 100 attributes. On the other hand, a device with a more advanced processor can compute the ciphertexts in around 700 ms even on the highest of the evaluated security levels, as Figure 6.4 shows. We can conclude, that devices with limited computational capabilities are not suited to encrypt data with high frequency. Since the ciphertext payload is independent of ABE with respect to execution time, we suggest that such a device should collect data over a period of time before encrypting it using ABE.

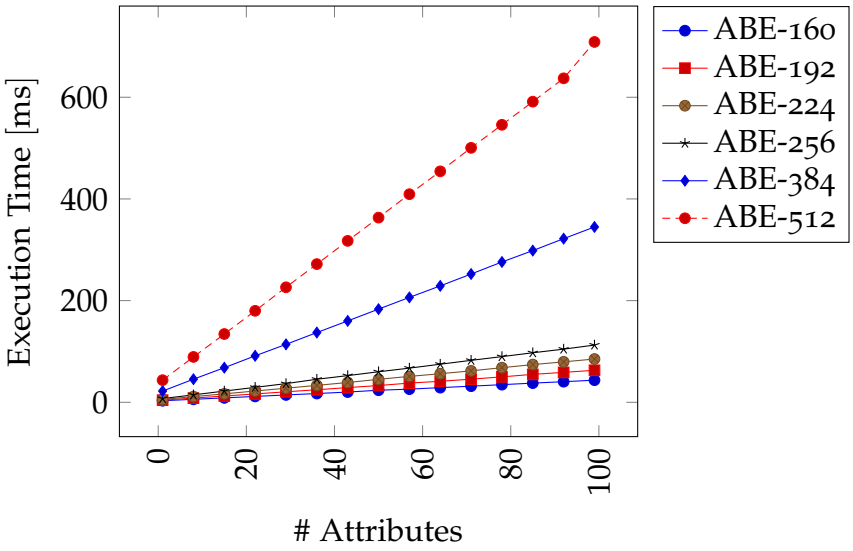

Figure 6.4.: Execution times of ABE encryption with prime order 160-bit to 512-bit. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz. From "Fine-Grained Access Control in Industrial Internet of Things: Evaluating Outsourced Attribute-Based Encryption" [65]

Re-Encryption When the RS receives a basic ciphertext, re-encryption based on attribute groups in $\mathcal{G}$ has to be performed. Equation (4.30) defines that each encrypted attribute C_t has to be re-encrypted by multiplying it with an attribute group key k_t. The computation involves a SHA-256 hashing and reduction in $\mathbb{Z}_p{}^*$ before multiplying each C_t with the corresponding k_t in $\mathbb{G}_1$. Creating all Hdr_t is therefore the main computational aspect of re-encryption. The RS generates two random values γ and μ once per re-encryption, which are needed to calculate all client elements x_k in the set of attribute groups $\mathcal{G}$. These are required for subsequent creation of the polynoms and resulting tuples protecting each k_t. The calculation of a client element x_k (see Equation (4.26) involves the application of H_0, H_T, a multiplication in each $\mathbb{G}_1$ and $\mathbb{G}_2$ as well as a bilinear pairing. The median execution time for calculating a x_k can be seen in Figure 6.5. Note that our prototype only calculates each x_k once, even if it is appearing in different groups. With high security levels this can become a significant factor, if there is a high amount of distinct clients associated with the affected attribute groups. The median execution time for determining polynomial factors

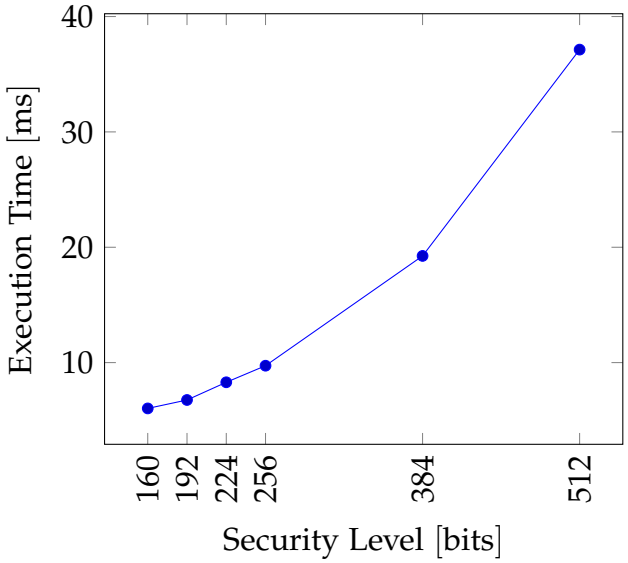

Figure 6.5.: Execution times for creation of client elements with prime order 160-bit to 512-bit. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz

from their roots is depicted in Figure 6.6.

These polynomial factors then have to be used to create the tuple, that is Hdr_t. Depending on the number of CLs in G_t, we have to perform $2 \cdot (|G_t| + 1)$ multiplications and one addition in $\mathbb{G}_1$, see Equation (4.28). The evaluation's [65] measurements on cloud-grade hardware for re-encryption in Figure 6.7 featured five clients per attribute group. We can see that high security parameters cause re-encryption to take seconds with increasing number of attributes. Full re-encryption only has to be performed on the initial upload of a basic ciphertext. However, updates in attribute groups have to be reflected in ciphertexts. In general, we showed in Section 4.2.7.2 that our chosen architecture only requires the affected Hdr_t to be rebuilt in such a case. Nevertheless, these changes affect all ciphertexts in a system that contain changed attribute groups. An option would be to lazily re-encrypt these ciphertext parts, the next time said ciphertext is requested by a CL. Anyhow, a fitting strategy needs to be devised based on the actual frequency of group membership changes. We leave this as an open question in this thesis.

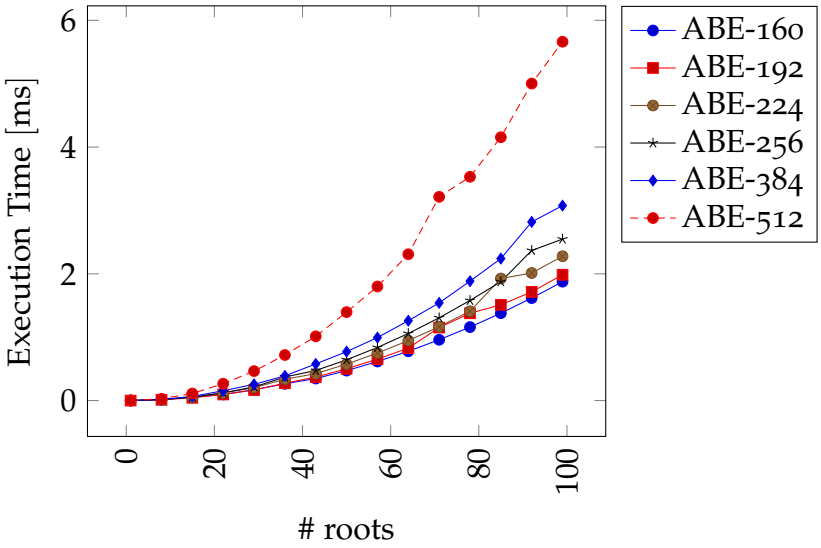

Figure 6.6.: Execution times for determining polynomial factors of an attribute group by its roots with prime order 160-bit to 512-bit, median of 100 iterations. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz

Partial Decryption We stated, that a dedicated service called DS has to perform partial decryption tasks. This service is invoked every time a client wants to recover plaintext in our system. The first step is the recovery of the attribute group encryption keys k_t. This involves the determination of the requesting CL's x and the calculation of a function based on the tuples Hdr_t, see Equation (4.32). If the CL is a member of that attribute group, the terms will cancel out and reveal k_t. This operation involves some exponentiations in $\mathbb{Z}_p{}^*$ and $|G_t|$ multiplications and additions respectively. Equation (4.33) shows, that the this step involves the recovery of the LSSS factors, see Appendix A.2 for an example. We can see in Figure 6.8 how the execution times develop with rising attribute count at different security levels.

However, the bulk of computational duties follows from Equation (4.34). We see, that for every ciphertext part C_t, we have to compute one multiplication in $\mathbb{G}_1$, two pairings as well as one multiplication and exponentiation in $\mathbb{G}_T$. The evaluation of the partial decryption performance in [65] showed, that it is indeed the most expensive operation in our ABE system. We can see in

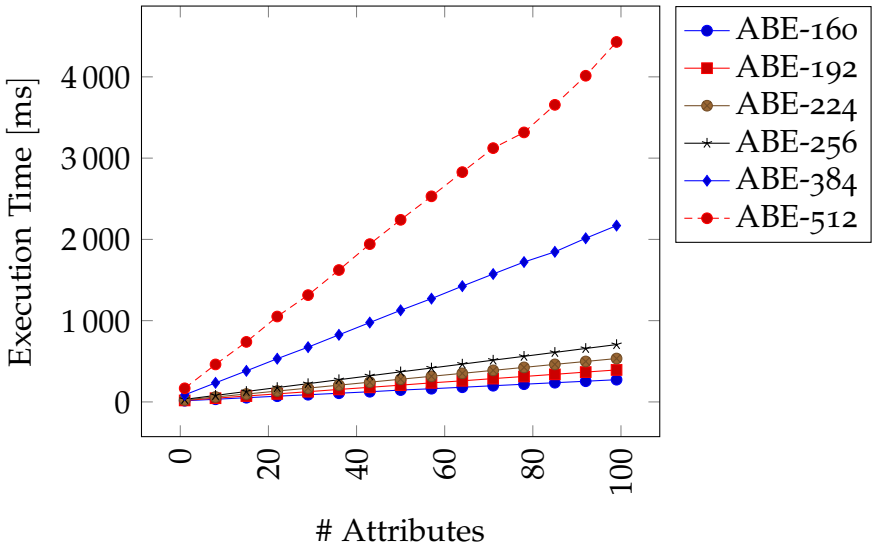

Figure 6.7.: Execution times of ABE re-encryption with prime order 160-bit to 512-bit. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz. From "Fine-Grained Access Control in Industrial Internet of Things: Evaluating Outsourced Attribute-Based Encryption" [65]

Figure 6.9, that at the highest security level with 100 attributes, our system needs around 6 seconds to partially decrypt a ciphertext.

Key Protocols Key creation and update operations depend on MPC. Equation (4.9) and Equation (4.14) are the common values KA and RS require to perform subsequent key calculations. The execution times to compute these initial values using *FRESCO* [19] can be seen in Figure 6.10. It shows, that the execution time is practically constant between 295 and 365 ms for the initial and update protocol respectively.

We defined the key protocols in Section 4.2.3.1 Section 4.2.3.2. As the private key-part of a KA incorporates the authorized attributes of the respective CL, its computation is dependent on the amount of attributes. Figure 6.11 shows the increased duration of the protocols with rising security parameters and increasing attribute count. The time it takes to update a KA key-part can reach multiple seconds on our testing system. We also see, that the timing for key-parts of the RS stay more or less constant, due to them being

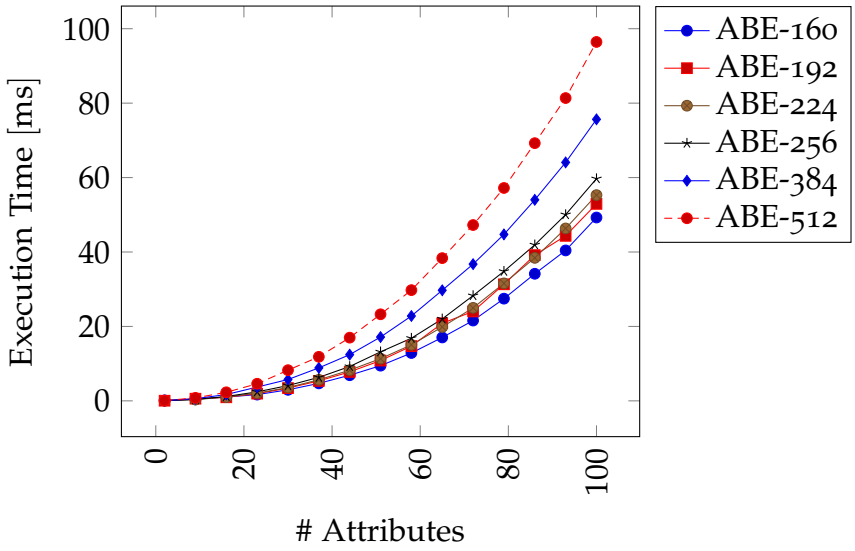

Figure 6.8.: Secret sharing recovery duration of ABE with prime order 160-bit to 512-bit. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz.

independent of attribute count. Since key creation and update should not occur frequently, the increased time it takes to compute the keys does not constitute a serious problem in an ABE system.

Serialization We examined the execution times of the core operations of our implementation in the previous paragraphs. Since our prototype's server components expose REST endpoints for communication, all interactions between actors in our ABE system requires data to be serialized to JSON. Therefore, we measured this computational overhead for conversion to and from JSON. We chose to examine basic ciphertexts and a KA's private keys, as they grow in size based on their associated attributes. We can see in Figure 6.12 that serialization to JSON is hardly measurable even with growing security parameter and large attribute sets. Deserialization on the other hand can take up to 80 ms in higher settings, because the group elements have to be explicitly decoded by *ECCelerate*^{TM}.

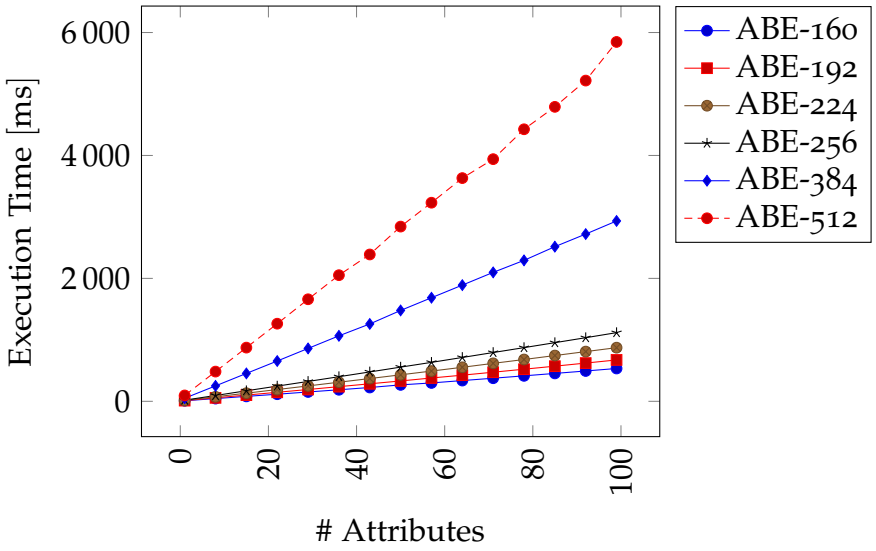

Figure 6.9.: Execution times of ABE partial decryption with prime order 160-bit to 512-bit. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz. From "Fine-Grained Access Control in Industrial Internet of Things: Evaluating Outsourced Attribute-Based Encryption" [65]

Concluding Remarks We see, that execution times of high-level ABE operations grow in a linear fashion, based on the amount of attributes involved. The secret sharing recovery uses Gauss-Jordan elimination, which was shown to have polynomial complexity by Farebrother [16]. The MPC execution time for initial values in key protocols, is only determined by how long it takes to setup and evaluate an MPC application with *FRESCO*. Interestingly, deserialization does take considerably longer compared to serialization, as serialized data has to be explicitly decoded with *ECCelerate*^{TM}.

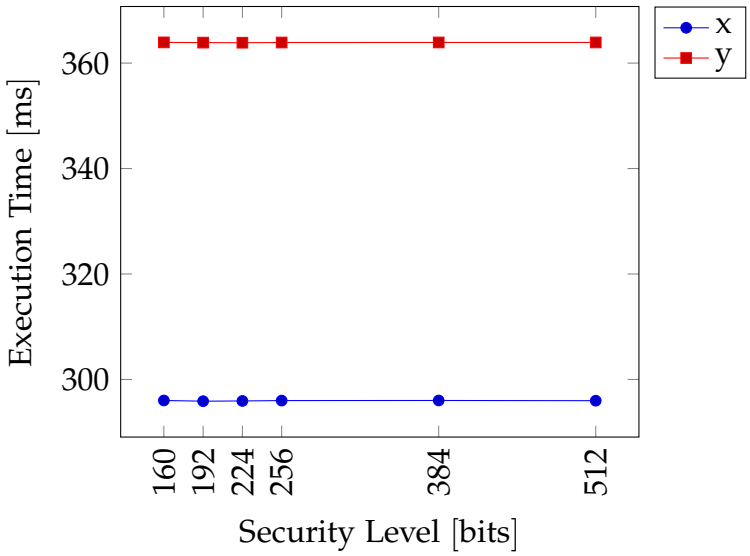

Figure 6.10.: Execution times for computing initial MPC values with prime order 160-bit to 512-bit. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz

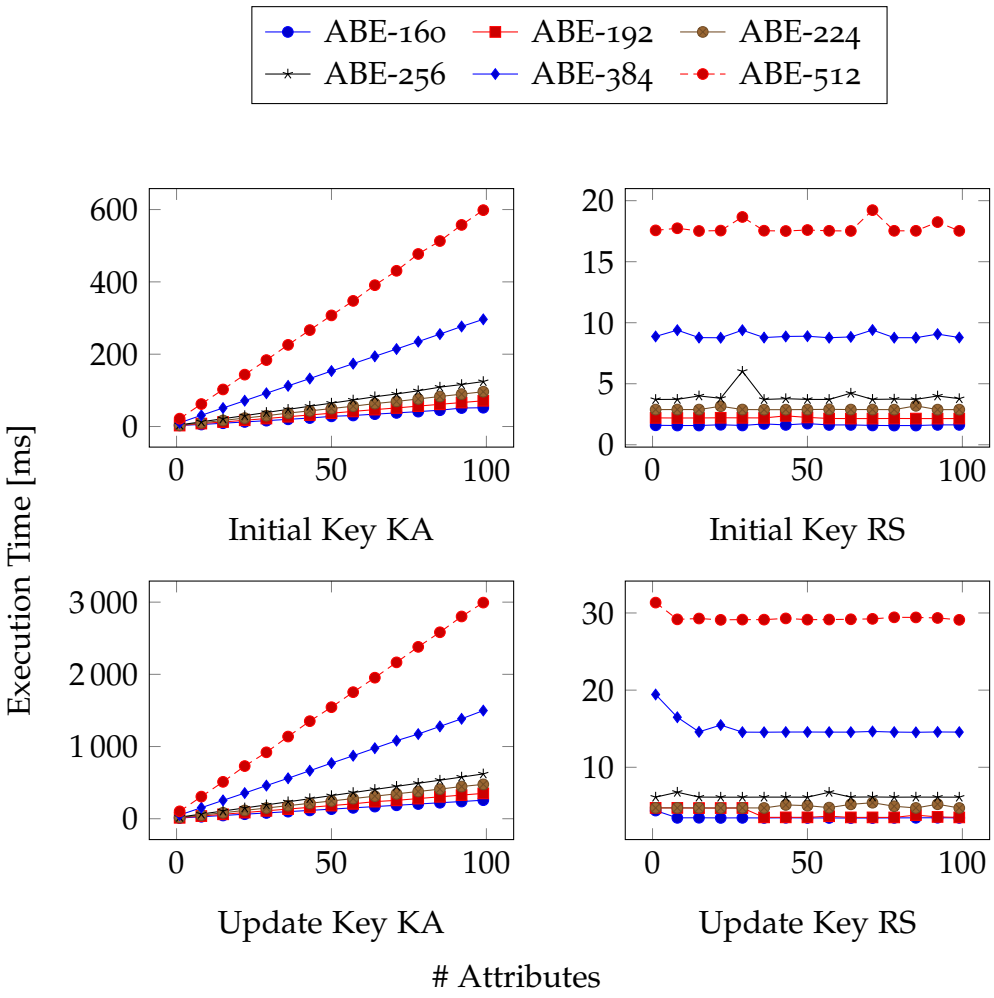

Figure 6.11.: Execution times of ABE key protocols with prime order 160-bit to 512-bit. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz.

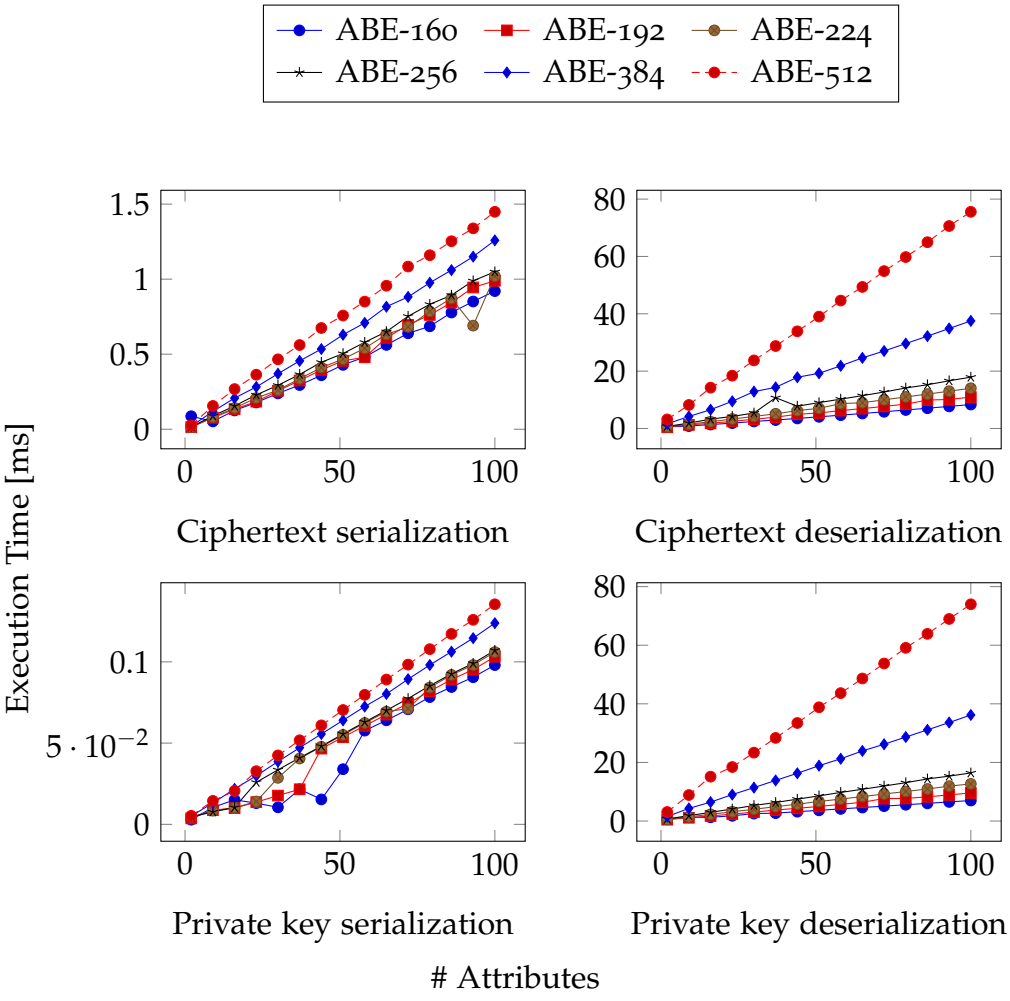

Figure 6.12.: Execution times of ABE serializations and deserializations with prime order 160-bit to 512-bit. Tests were executed on an Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz.

7. Conclusion

The goal of this thesis was to demonstrate that Attribute-Based Encryption (ABE) can be a suitable mechanism to provide data access control on a cryptographic level. Our system targeted the Industrial Internet of Things (IIoT), a heterogeneous environment of devices with different computational resources. The mathematical foundation of our system is based on the work of Lin, Hong, and Sun [38]. We adapted their architecture to work with bilinear pairings in the asymmetric Type III setting. Using the Ate pairing over Barreto-Naehrig (BN) curves, we achieve better performance and higher security compared to symmetric bilinear pairings over supersingular curves.

Our implementation was evaluated with regards to ciphertext size and execution performance. We extend results from the evaluation by Ziegler, Sabongui, and Palfinger [65], by providing measurements for the building blocks for our cryptographic functionality. With security parameters ranging from 160 to 512 bit, we analyzed how our system performs with respect to near-, mid- and long-term security. We see that the amount of attributes in access structures directly influences execution time and data overhead. The most expensive operations in our system are re-encryption and partial decryption, which are targeted to be performed by cloud services with considerable computing power. These operations only take fractions of a second for security parameters of 256 bit and below. When targeting both long term security and a high numbers of attributes, these procedures can take as long as 5-6 seconds. Other operations with comparable execution times are the key creation and update protocols. However, in these cases the long duration isn't critical as they happen infrequently. Basic encryption with 100 attributes can be performed in about 100ms for up to 256 bit security and taking about 700 ms on the highest of our tested security levels. As this operation would also be targeted towards IIoT endpoints,

we additionally evaluated it on a low performance device. We saw that a Raspberry Pi 3 could perform encryption using 256 bit security with up to 20 attributes in about one second. We conclude, that such devices should not perform this operation with high frequency or at long-term security levels. Buffering data before encryption is always recommended, as ABE adds considerable overhead to ciphertexts. Another option for computationally restricted equipment would be to send data to an intermediate and more powerful device for subsequent encryption with ABE. The final decryption step is independent of attributes and always has constant execution time. A Raspberry is able to perform decryption in about 150 ms with 256-bit security. More potent endpoints only require 5-30 ms on different security levels.

7.1. Future Work

While we presented a prototype that can achieve reasonable performance on various devices, we also identified several areas that would benefit from additional improvements.

Linear Algebra When we started to work on the implementation of our system, no suitable linear algebra library existed for our operations in $\mathbb{Z}_p{}^*$. Therefore, we created our own matrix (including attribute labels) and Gauss-Jordan elimination implementations. Even though we showed that these operations only constitute for a fraction of the overall computational efforts, it would be interesting to see if more mature linear algebra libraries providing this functionality would improve performance. Especially in the context of secret sharing factor recovery.

Ciphertext Size As we lined out in the previous chapters, our ciphertexts include the full share matrix M_{share} which does contribute significantly to the ciphertext size. Using our current approach, a Decryption Server (DS) will always be able to correctly recover Linear Secret Sharing Scheme (LSSS) factors for authorized Clients (CLs). A possible performance improvement

could be to send the policy string instead, as it requires far less representative data. Subsequently, a DS would have to build the matrix on every decryption request. However, the computational overhead stays in a lower single digit millisecond range, as we showed in Figure 6.2. In this case, we would have to ensure that updated secret sharing implementations produce identical matrices. Otherwise, old ciphertexts would not be decryptable. Additionally, alternative ways to create LSSSs and Monotone Span Programs (MSPs) could also be considered. Liu and Cao [39] presented a way to create policies based on threshold gates that reduces the amount of rows in a resulting matrix.

Computational Performance Chapter 6 showed, that our implementation's most expensive operations can take as long as several seconds with large security parameters and high attribute count. A possible speedup of pairing operations can be achieved, through simultaneously evaluating multiple pairings that have a fixed element in $\mathbb{G}_2$, as Ramacher [46] points out. This is actually the case for pairings in our re-encryption and partial decryption process, as can be seen in Equation (4.26b), Equation (4.31b) and Equation (4.34). So an adaption of our implementation in this regard is definitely possible.

7.2. Outlook

We showed that ABE is a viable option for enforcing data access control in an environment like IIoT. Deployment on computationally restricted devices remains a challenge with regards to encryption, as they start to struggle in ABE systems with high security parameters. Possible solutions to explore are intermediary systems for encryption or endpoints equipped with hardware that can efficiently compute bilinear pairings. Strategies for updating ciphertexts due to attribute group changes should be investigated, as naive approaches are not effective in systems with large amounts of stored ciphertexts. Going forward, alternative ways to compute and distribute private key-parts are also worth exploring.

Appendix A.

Algorithms and Examples

A.1. Building a Polynomial from its Roots

The ABE system builds polynomials to determine if a CL belongs to an attribute group. In this case, the CLs' associated elements $x_k \in \mathbb{Z}_p{}^* \mid u_k \in G_t$ are the roots of the desired polynomial. This means that whenever the polynomial is evaluated for an authorized x_k, the result is 0. This property is necessary for the DS to successfully recover an attribute group key (see Equation (4.32)). The polynomial's definition (see Equation (4.27)) states, that all terms in the form of $(x - x_i)$ have to be multiplied. For an example set of roots $(1, -2, 3)$ this would mean:

$$\begin{aligned} f_t(x) &= \prod_{i=1}^{v} (x - x_i) = (x-1)(x+2)(x-3) \\ &= \left(x^2 + x - 2\right)(x-3) \\ &= x^3 - 2x^2 - 5x + 6 \end{aligned} \tag{A.1}$$

To programmaticaly calculate this, this post [1] presented a pseudocode. The actual implementation in the prototype slightly adapted the computation to account for indices starting at 0 and all calculations happening in $\mathbb{Z}_p{}^*$ (for example $p = 17$):

[1] `https://stackoverflow.com/a/32932482`

Listing A.1: Calculating a polynomial from its roots

```
ZpPolynomial resultPolynomial = new ZpPolynomial(roots.length, p);

BigInteger[] result = new BigInteger[roots.length+1];
result[0] = BigInteger.ONE;
Log.debug("Creating polynomials from roots {} (mod {})", roots, p);
Log.debug("Result array at start: {}", (Object[])result);

for (int n = 0; n < roots.length; n++) {

  BigInteger value = roots[n].multiply(result[n]);
  value = BigInteger.valueOf(-1).multiply(value);
  result[n+1] = value.mod(p);
  Log.debug("Intermediate result (n={}): {}", n, result);

  for (int k = n; k >= 1; k--) {
    BigInteger update = result[k].subtract(roots[n].multiply(result[k
    -1]));
    result[k] = update.mod(p);
    Log.debug("Intermediate result (n={}, k={}): {}", n, k, result);
  }

  Log.debug("Result (n={}): {}", n, result);
}

int degree = resultPolynomial.getDegree();
for (int i = 0; i <= degree; i++) {
  resultPolynomial.setCoefficient(degree-i, result[i]);
}

Log.debug("Created polynomial from roots {}: {}", roots,
    resultPolynomial);

return resultPolynomial;
```

The final loop for actually filling the `ZpPolynomial` object is necessary, as the indexing works different than in the `BigInteger` array. While the coefficient

of the highest power is found at index 0 in the array, it would be retrieved via the `getCoefficient(int power)` method of the `ZpPolynomial` object. The resulting output of this code snippet would be the following:

Listing A.2: Output of the calculation of a polynomial

```
Result array at start: 1
Intermediate result (n=0): [1, 16, null, null]
Result (n=0): [1, 16, null, null]
Intermediate result (n=1): [1, 16, 15, null]
Intermediate result (n=1, k=1): [1, 1, 15, null]
Result (n=1): [1, 1, 15, null]
Intermediate result (n=2): [1, 1, 15, 6]
Intermediate result (n=2, k=2): [1, 1, 12, 6]
Intermediate result (n=2, k=1): [1, 15, 12, 6]
Result (n=2): [1, 15, 12, 6]
Created polynomial from roots [1, -2, 3]: 1*x^3 + 15*x^2 + 12*x
    ^1 + 6*x^0 (mod 17)
```

A.2. Creating a MSP and recovery of factors through solving linear equations

Take the example policy in Figure A.1, in the notation used in the prototype:

Listing A.3: Policy string in postfix notation

```
1 ( E, ( (A, B, OR), ( C, D, OR), AND ), AND )
```

When creating a MSP, one has to decide what the sharing vector will be. Using the method presented in [37], the sharing vector will be $(1, 0, \cdots, 0)$. Operating in $\mathbb{Z}_p{}^*$ with $p = 17$, the resulting sharing matrix $\mathcal{M}$ for this policy is:

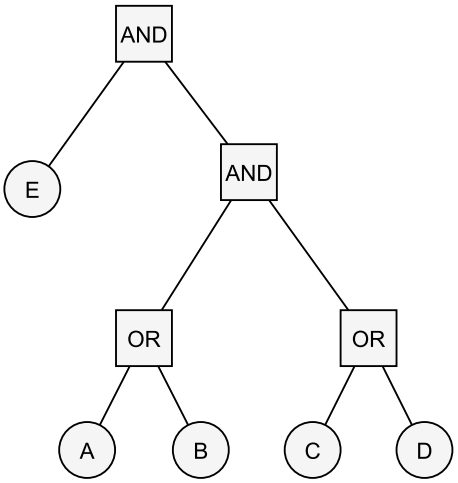

Figure A.1.: Graph of an access policy example

$$\mathcal{M} = \begin{matrix}(E)\\(A)\\(B)\\(C)\\(D)\end{matrix}\begin{pmatrix}1 & 1 & 0\\0 & 16 & 1\\0 & 16 & 1\\0 & 0 & 16\\0 & 0 & 16\end{pmatrix} \tag{A.2}$$

Using Equation (2.4), we get the vector containing the shares. They correspond to participants, as is shown in the labeling of Figure A.1. Take $s = 3$ as a secret and use it as the first element of an otherwise random vector $v = (3, 2, 6)$. When multiplying it with the sharing matrix $\mathcal{M}$, we get the resulting vector v_s.

$$v_s = \mathcal{M} \cdot v^T = \begin{matrix}(E)\\(A)\\(B)\\(C)\\(D)\end{matrix}\begin{pmatrix}5\\4\\4\\11\\11\end{pmatrix} \tag{A.3}$$

The shares would be distributed to participants according to their labeling. When a set of participants wants to recover the secret using only their shares, we have to recover the according factors (see Equation (2.5)).

Recovery is possible by solving a system of linear equations, based on the sharing matrix. First, we only take those rows of $\mathcal{M}$ with labels corresponding to the recovering set of participants. With $\{A, C, E\}$ as an example set of participants, we obtain the submatrix $\mathcal{M}'$ and a vector containing the relevant shares:

$$\mathcal{M}' = \begin{matrix}(E)\\(A)\\(C)\end{matrix}\begin{pmatrix}1 & 1 & 0\\0 & 16 & 1\\0 & 0 & 16\end{pmatrix}, v'_s = \begin{matrix}(E)\\(A)\\(C)\end{matrix}\begin{pmatrix}5\\4\\11\end{pmatrix} \tag{A.4}$$

A set of participants fulfills the underlying access policy iff their matrix rows are a linear combination of the sharing vector. Therefore, the next step would be to transpose and augment the matrix $\mathcal{M}'$ (without labels) with the previously defined sharing vector. In our example this would result in the following augmented matrix:

$$\mathcal{M}'_{aug} = \left(\begin{array}{ccc|c}1 & 0 & 0 & 1\\1 & 16 & 0 & 0\\0 & 1 & 16 & 0\end{array}\right) \tag{A.5}$$

We want to recover the factors that let us combine the participant rows to build the sharing vector. The augmented matrix now represents a set of linear equations that we want to solve. If there exists a solution, we can bring the augmented matrix into the so called Reduced Row-Echelon Form (RREF). If it is possible to achieve this, the values to the right of the vertical separator are the factors that we need to apply to the participant vectors to get the linear combination of the sharing vector.

The RREF of the matrix can be obtained through Gauss-Jordan elimination[2]. The augmented matrix of our example would be solved through the following steps:

[2] http://www.math.jhu.edu/~bernstein/math201/RREF.pdf

$$\begin{aligned}\mathcal{M}'_{aug} = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 1 & 16 & 0 & 0 \\ 0 & 1 & 16 & 0 \end{array}\right) &\rightarrow \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 16 & 0 & 16 \\ 0 & 1 & 16 & 0 \end{array}\right) \\ \rightarrow \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 16 & 0 \end{array}\right) &\rightarrow \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 16 & 16 \end{array}\right) \\ \rightarrow \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{array}\right)\end{aligned} \tag{A.6}$$

We can see, that this easy example is solvable. The sharing vector is a linear combination of the participant vectors, with all factors being 1. At last, we will multiply a vector containing the recovered factors v_r with v_s':

$$v_r \cdot v'_s = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 4 \\ 11 \end{pmatrix} = 3 \tag{A.7}$$

Now we show an example set of participants that don't fulfill the access policy, $\{A, B, E\}$. We begin by defining $\mathcal{M}'$ and v_s':

$$\mathcal{M}' = \begin{matrix} (E) \\ (A) \\ (B) \end{matrix} \begin{pmatrix} 1 & 1 & 0 \\ 0 & 16 & 1 \\ 0 & 16 & 1 \end{pmatrix}, v'_s = \begin{matrix} (E) \\ (A) \\ (B) \end{matrix} \begin{pmatrix} 5 \\ 4 \\ 4 \end{pmatrix} \tag{A.8}$$

We then try to solve the system of linear equations:

$$\begin{aligned}\mathcal{M}'_{aug} = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 1 & 16 & 16 & 0 \\ 0 & 1 & 1 & 0 \end{array}\right) &\rightarrow \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 16 & 16 & 16 \\ 0 & 1 & 16 & 0 \end{array}\right) \\ \rightarrow \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{array}\right) &\rightarrow \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 16 \end{array}\right)\end{aligned} \tag{A.9}$$

We see in the last row, that we cannot solve this system. This is the result of the participant rows in $\mathcal{M}'$ not being a linear combination of the sharing vector.

Appendix B.

REST APIs

The following sections list all exposed Representational State Transfer (REST) endpoints for the server components of the ABE prototype.

B.1. abe-server-keyauthority

/rest/parameters/base	
Actor	StS
Method	GET
Parameters	System-UUID
Body	
Summary	Retrieves the initially generated system parameters.
Response	see Listing B.1

Listing B.1: Example of a base system parameter result

```
{
  "dataIntegrityInformation" : {
    "mode" : "SHA256",
    "value" : "9CYVGx2QYoCYfGlpjkCd0dDVkysUvyMN5g+aynlaNy8=",
    "salt" : "pAKVXbuGyFnhB6w+zywp9tXHC5QA8SanCMxVk0f0niQ="
  },
  "security" : 160,
  "hashType" : "SIMPLE_SHA256",
  "systemAttributes" : {
    "factory" : "AiZbgqkSyptCemlamsc9g2YUMQL2",
```

```
    "operations" : "Ayucp3ZH6/5oa5s0y9vUwAWI8CNN",
    "management" : "A5n2rmcqyiuN0cKrY5I6QJgXWn+c",
    "purchasing" : "AiND3N2tntzfBWCxJ1NY334FnC+C",
    "quality" : "Ay8SBarPCVtKEMdLEw9xMhp8pFlh"
  }
}
```

/rest/parameters/system	
Actor	StS, DS, CL, DO
Method	GET
Parameters	System-UUID
Body	
Summary	Retrieves the full system parameters.
Response	see Listing B.2

Listing B.2: Example of a system parameter result

```
{
  "dataIntegrityInformation" : {
    "mode" : "SHA256",
    "value" : "zAcJNUxTCDS3KR8nNptegKP70uKQYZDCQNWN0KQyqWY=",
    "salt" : "Za2nbGVVs/aokFSyEFNrpfHNm6sOWLPj6ia0afLSAcQ="
  },
  "keyAuthorityPublicParameter" : "AzRA3KwNZTs4qNEj037F1BK2IUvZ",
  "reencryptionPublicParameter" : "J0mstBXT07kRLYdDI3gRwbJFiiyjobV3p1lILCLjjyJ
    O8V0p45FjUYHS2qc1hnRT7GLYIVVPxmssdjWlQ1+ZPytfQ3Pd0Mb72cxLo+gW7dBt2cVivVQup
    /zQrgwPIQn2+NP1eycvKaTf7/+VpyYgn6giRx/7OlOgSjY5U4ASQbW27DI2s7J3CcFogHsntZk
    NgdmwjhHLAXLSOqCPptWSBYPr3WuKxJ/uFfF4n7nctL9EXRS4pySP8mgAYygk0Pg90ndau39aF
    /JnsByZIKFhMx5Vk2OyhNvGSeMpUKL9t925EOUL5vYoab6IVLmy81Q2",
  "baseSystemParameters" : {
    "security" : 160,
    "hashType" : "SIMPLE_SHA256",
    "systemAttributes" : {
      "factory" : "AiZbgqkSyptCemlamsc9g2YUMQL2",
      "operations" : "Ayucp3ZH6/5oa5s0y9vUwAWI8CNN",
      "management" : "A5n2rmcqyiuN0cKrY5I6QJgXWn+c",
      "purchasing" : "AiND3N2tntzfBWCxJ1NY334FnC+C",
      "quality" : "Ay8SBarPCVtKEMdLEw9xMhp8pFlh"
    }
  }
}
```

/rest/parameters/system/register	
Actor	StS
Method	POST
Parameters	System-UUID
Body	Public re-rencryption parameters, see Listing B.3
Summary	Registers the re-rencryption parameters at the KA to finish the system setup and returns the finalized ABE system parameters
Response	see Listing B.2

Listing B.3: Example body of a POST request for registering the public reencryption parameter.

```
{
  "dataIntegrityInformation" : {
    "mode" : "SHA256",
    "value" : "XcOVKUVsxK6u6nTuS4ScpwolOWvyaQ0PiBDI5hqFu9Q=",
    "salt" : "mVV/WHio95czE9rQx1X0eghmBwNfxpNmkhMIxC9DDj8="
  },
  "publicParameter" : "kz5rmxYAZoYqStEE6dTHyyi+xKbKL4KLVI7lYTxoZBQ56gWVv1n64HZ
    A1dtatU3+CVjX7F+zfeXjunvdJRPgk3Ya5wO8OVNjGtH3hhRZamkzVPwmIlI7hbJSnOx52TRm4
    JKBdyjMrW0o5ITZ4kh0Smue2CpxiCBRmPdHF/FSLE/KkKZ7OT1l3i6syftnTCabKAwz45oLRJk
    V4WysVSkn1mRCt7UT2EH1RSgiU7vpHnWlu8Djwq1jjarZMRxUCdWcz+VRYbmLRqaDoIv7uCVWp
    Ub2Uvc3lHXzL87zHI2fEGVGmYN153R9RU0wR9bYVKag"
}
```

/rest/parameters/groups/all	
Actor	StS, DS
Method	GET
Parameters	System-UUID
Body	
Summary	Retrieves the entire set of attribute groups from the system.
Response	see Listing B.4

Listing B.4: Example of an attribute group result

```
[ {
  "attribute" : "factory",
```

```
  "clients" : [ "Francis Overwood", "Fitzgerald Edmund", "Frida" ]
}, {
  "attribute" : "management",
  "clients" : [ "Margaret CFO", "Michael CTO", "Mark CEO", "Superuser" ]
}, {
  "attribute" : "operations",
  "clients" : [ "Olga Kurolinko", "Oliver Stane", "Orbun Viktor", "Superuser"
    ]
}, {
  "attribute" : "quality",
  "clients" : [ "Quincy Jonas", "Quinn Dr.", "Quentin Tarantula", "Superuser"
    ]
}, {
  "attribute" : "purchasing",
  "clients" : [ "Penultimo Tropico", "Patrick Daffey", "Pamela Andersdottir" ]
} ]
```

/rest/parameters/groups/relevant	
Actor	StS, DS
Method	GET
Parameters	System-UUID
Body	A set of system attributes
Summary	Retrieves the attribute groups associated with the provided system attributes.
Response	Returns a subset of Listing B.4

/rest/client/key/new	
Actor	CL
Method	GET
Parameters	System-UUID, User-ID
Body	
Summary	Initiates a key update for a user. Returns a UUID associated with the MPC computation.
Response	UUID of an MPC associated with this request

/rest/decryption/key	
Actor	DS
Method	GET
Parameters	System-UUID, User-ID
Body	Private key-part
Summary	Retrieves the KA's private key-part for a user.
Response	see Listing B.5

Listing B.5: Example of a KA's private key part for a given user

```
{
  "dataIntegrityInformation" : {
    "mode" : "SHA256",
    "value" : "89+JfnSpNOgXUZxZMfyDMHG2ztv8qyqdvedGJA2p4VA=",
    "salt" : "HCNO6FDJdx3hDmmY0U67gXN2YI+JYhHon1+1jyEsDQA="
  },
  "privatePart" : "A0jdiuSdUQrbo6zUS+got9TW/1EuemmKXR92Ybs97XGXqzwSPIc5eOo=",
  "attributes" : {
    "test2" : "A1ZpCsLDkGWPTUhgHLG777NxUZX6",
    "test3" : "AxcKOhMTAazHBhkmqAW8ToMLFlmO",
    "test1" : "AiAaJdeNu0vGqSjTX8ihcL+jykzl"
  }
}
```

B.2. abe-server-storage

/rest/key/init	
Actor	KA
Method	GET
Parameters	System-UUID, User-ID, MPC-Calc-ID
Body	
Summary	Notification from the KA that a new key should be created for a user with the given MPC UUID.
Response	Returns an HTTP error code on an unsuccessful call

/rest/key/invalidate	
Actor	KA
Method	GET
Parameters	System-UUID, User-ID, MPC-Calc-ID
Body	
Summary	Notification from the KA that a user's key should be immediately invalidated.
Response	Returns an HTTP error code on an unsuccessful call

/rest/key/update	
Actor	KA
Method	GET
Parameters	System-UUID, User-ID, MPC-Calc-ID
Body	
Summary	Notification from the KA that a user's key should be updated with the given MPC UUID.
Response	Returns an HTTP error code on an unsuccessful call

/rest/decryption/key	
Actor	DS
Method	GET
Parameters	System-UUID, User-ID, MPC-Calc-ID
Body	
Summary	Retrieves the StS's private key-part for a user.
Response	see Listing B.6

Listing B.6: Example of a Re-Encryption Server (RS)'s private key part for a given user

```
{
  "privatePart" : "A5/By2K82dAffUZCmi61L7ovntaC"
}
```

/rest/ciphertext/upload	
Actor	DO
Method	POST
Parameters	System-UUID, Ciphertext-Path
Body	A basic ciphertext to be re-encrypted, see Listing B.7
Summary	Uploads a new basic ciphertext and returns the unique UUID.
Response	The UUID of the uploaded ciphertext

Listing B.7: Example of a basic ciphertext to be uploaded.

```
{
  "shareMatrix" : "[1, 1, 0, 0](management)\n[0, 123736091430160831025720604254654
    53821169514836108, 1, 0](operations)\n[0, 0, 123736091430160831025720604254654
    53821169514836108, 0](quality)\n[0, 123736091430160831025720604254654
    53821169514836108, 0, 1](purchasing)\n[0, 0, 0, 123736091430160831025720604254654
    53821169514836108](factory)\n(mod 123736091430160831025720604255465382
    1169514836109)",
  "encryptedElement" : "hXFMq4V/zkKhqgmiW0zrz553vayW3uekyEw1Awxv0I/8oYfvkNlR2m
    cQihnLVdSjFPzM2LraODzC/Tb7toPsnlRlaeXsajEExm6/SJq78zJ7krEwCEI0S7mUgHVExe0y
    u5Blw0DlrqRrZLeYByyKToSMXVMl/AQ/CR0kdaVwVBe39XXBlfL1PH6QpLyLorauq9aUNao//6
    W0zffgDYn+L1z9vpDNFdNnqJ7tkdTBlJHexyRBybWoqwHlqGJHQ8tYGltApELlzi+yljBdY7Pk
    3Lyu215OwF/BrwxXDdSnzzbGRlIE82tFCCthZgcO0vZa",
  "staticPart" : "A4zVI2s44TeWWSVf3ca0LXh0VwgkngASnMtp0EVkYEO8VuuNew8G9os=",
  "encryptedAttributes" : [ {
    "label" : "management",
    "point" : "AlLXiepn3hPBpfp86BTIYxSxS6Ec"
  }, {
    "label" : "operations",
    "point" : "A4AjfJlQ6uqT4GMchy6dAbRy8FKZ"
  }, {
    "label" : "quality",
    "point" : "An45WYX7SgOguFs5JtNM0SM6RF1A"
  }, {
    "label" : "purchasing",
    "point" : "AlMTUxkvBM1EXXcvxC4171Rn7iDF"
  }, {
    "label" : "factory",
    "point" : "ApWHmas3+7Q6++jPfFCQ6Xsm9GA5"
  } ],
  "encryptedPayload" : {
    "iv" : "FAqunIIHSjtbtxMzFJXACQ==",
```

```
    "payload" : "p9bYVaM9AqCaunbu0CubJwfnUhZKz2SDmhfwsxcJG8KEmSjAiNAmRA2QV6r88
    1EzMhgm6jraOZIgivdvvwsaNOnZ0QFlTbVW/fPy3xwkC3VuCNIvHJCh3bVfhDn7spPV0m92CSX
    X++93N+6wjM8jxw=="
  }
}
```

/rest/ciphertext/retrieve	
Actor	CL
Method	GET
Parameters	Ciphertext-UUID
Body	
Summary	Retrieves a ciphertext with the given UUID
Response	see Listing B.8

Listing B.8: Example of a full ciphertext

```
{
  "header" : {
    "part1" : "A5/NrXFOlUpvDB9J+vIqsH1LDtoFPQ1Dvm6+VuyjsuQgNKgejbueoNc=",
    "part2" : "EPkfhe0lT2VykEhm2frfZBsZOw4=",
    "groupHeaders" : [ {
      "groupName" : "management",
      "groupTuple" : [ "AjcuRGso0R0mx+YsUTcFeRYx01/4", "AgUmYLMfWaeqhk7CTxN2eZ
    slF6LJ", "AhlG2f3Vaz/w9lQFde4c0WkmGbZz", "A5bmekHYMaAaGXoybg9AVHxmYRHg", "
    Ay1wKhCg/7rp0DlVxfIf6SkpwDhP" ]
    }, {
      "groupName" : "operations",
      "groupTuple" : [ "As+iYVsNp66Wo5fkpuCZkIddiBas", "A9P2A8rf/95OsZf/CHNtXo
    Ifu1HT", "A4C8J82vtfLC/xOKUh6NuLo7JE5B", "A8wizgGDk/P8M4CXW/lRqb948KsC" ]
    }, {
      "groupName" : "quality",
      "groupTuple" : [ "AireZIwfeSq8advMpdGH9i6ibaHc", "AnyP01USTyX1MxA3ftI0F6
    jQS3L4", "Aq8KnAVEtZToRGc+ORVpSYJR8nHL", "A80TJuhdqV2zxKl80PQTq7m7khC0", "
    AnvxU5/qnzNA22IIEYY4wrqE/Zyn" ]
    }, {
      "groupName" : "purchasing",
      "groupTuple" : [ "AoBy9iSQ6yjuPqpdnaJwR3vINXki", "A4zuBG3BNHtF660lmAAorT
    S437jO", "AoxGcMQ/jAib52H7sswAOz1/bhyt", "Ah+JhNJyXxhG1ObNKFOhOi6uSBuD" ]
    }, {
      "groupName" : "factory",
```

```
      "groupTuple" : [ "AnKC44HTMHORatlaPZDBFLBQrs+k", "Ah/4waqkLQ/OYUs7IgOUJ7
    BcBQg2", "A3UqDoWPth3cvOjTxe2GTHhmoEHU", "AoLWOflE4dqnO12Gx6/wE8qvxfZ5" ]
    } ]
  },
  "basicCiphertext" : {
    "shareMatrix" : "[1, 1, 0, 0](management)\n[0, 123736091430160831025720604
    2554653821169514836108, 1, 0](operations)\n[0, 0, 12373609143016083102572 0
    60425546538211695148361 08, 0](quality)\n[0, 123736091430160831025720604255
    4653821169514836108, 0, 1](purchasing)\n[0, 0, 0, 12373609143016083102572 0
    60425546538211695148361 08](factory)\n(mod 123736091430160831025720604255 46
    53821169514836109)",
    "encryptedElement" : "hXFMq4V/zkKhqgmiW0zrz553vayW3uekyEw1Awxv0I/8oYfvkNlR
    2mcQihnLVdSjFPzM2LraODzC/Tb7toPsnlRlaeXsajEExm6/SJq78zJ7krEwCEI0S7mUgHVExe
    0yu5Blw0DlrqRrZLeYByyKToSMXVMl/AQ/CR0kdaVwVBe39XXBlfL1PH6QpLyLorauq9aUNao
    //6W0zffgDYn+L1z9vpDNFdNnqJ7tkdTBlJHexyRBybWoqwHlqGJHQ8tYGltApELlzi+yljBdY
    7Pk3Lyu215OwF/BrwxXDdSnzzbGRlIE82tFCCthZgcO0vZa",
    "staticPart" : "A4zVI2s44TeWWSVf3ca0LXh0VwgkngASnMtp0EVkYEO8VuuNew8G9os=",
    "encryptedAttributes" : [ {
      "label" : "management",
      "point" : "A9DVLmOugsOdh0VHfKJ841yYh4la"
    }, {
      "label" : "operations",
      "point" : "AjkbBajCoL5l/OG4bmxXSyOZFvfU"
    }, {
      "label" : "quality",
      "point" : "AzM+JG24dX4VL3r7SfTGSAOu0JiG"
    }, {
      "label" : "purchasing",
      "point" : "Ars2zZBEp6Pw2hTOIhLCXBeVTsTb"
    }, {
      "label" : "factory",
      "point" : "Ahok5pUcrq9bxA+rk5fqgaBknVB5"
    } ],
    "encryptedPayload" : {
      "iv" : "FAqunIIHSjtbtxMzFJXACQ==",
      "payload" : "p9bYVaM9AqCaunbu0CubJwfnUhZKz2SDmhfwsxcJG8KEmSjAiNAmRA2QV6r
    881EzMhgm6jraOZIgivdvvwsaNOnZ0QFlTbVW/fPy3xwkC3VuCNIvHJCh3bVfhDn7spPV0m92C
    SXX++93N+6wjM8jxw=="
    }
  }
}
```

/rest/ciphertext/list	
Actor	CL
Method	GET
Parameters	System-UUID, Ciphertext-Path, Ciphertext-UUID(optional)
Body	
Summary	Retrieves a list of ciphertexts that were associated with a given path. If Ciphertext-UUID is supplied, only ciphertexts that were stored after that point in time will be returned.
Response	A list of ciphertexts

/rest/ciphertext/path/newest	
Actor	CL
Method	GET
Parameters	System-UUID, Ciphertext-Path
Body	
Summary	Returns the last ciphertext UUID that was associated with the given path.
Response	Returns the most recently generated UUID for a given ciphertext path

B.3. abe-server-decryption

/rest/decrypt	
Actor	CL
Method	GET
Parameters	System-UUID, User-ID
Body	A re-encrypted ciphertext
Summary	Partially decrypt a given ciphertext for a given user.
Response	see Listing B.9

Listing B.9: Example of a partially decrypted ciphertext.

```
{
```

```
  "encryptedElement" : "JndONgLF5oEWAL9yUpqU805Z2iNiwgyzfdZjIbqkZJO0qWM5iq5d+Y
    8Smst3koQ+ZLaBs9qCA6BUOR9oDKfW5hdPhw4FRl42sW3eFLBjZ1Y/AONGO1ajiFCuzqsoUjBk
    8kiXXgE6mZIpx/jUUUTjmzP0Tca/4+oBF3G78rCrC62K/frlgKjVtLsIVNM5/85MfW7/BYeVC4
    t/jE33tEUQvSoIa4WUu8VB/RjeGJFFm7dy27cEOuvc1vSea+n3uAMx4dxg2BiMJSB01y9eTvgj
    8mnUcHLi37V+v9Jv4A63cE8/Jh2faMKSH0H4sOr+m93x",
  "partiallyDecryptedElement" : "pFmbp2/B6BVCMnL3tMDd0XhWQlcR2AF1CoHnn7IaFFeg5
    VpV57IuQD32VKG7yyak/CYm93xe1UCIFwxwFrX+rq2P5MdVPDpIsK6KpsE3y1h4hHfdF07+uk+
    WmImZ1poRUJFbqthIkDyQShGq3e3tOTa0xfelgRcZABJMm9lrjByJl+42UCRYGkTZ8M9dH47BP
    oUDBWK34UKdEeirtSNhmKfIxdl60PmXedfOxfl4kqeI3OSWQpl/fIC4PuxkK1ID0fdnfji0tUm
    5L3HC0deXFlQOrmtbFB9kG6gEy1mZCCyT81QZssgRaCVpSX7DGBaY",
  "encryptedPayload" : {
    "iv" : "dw2F2AVLDuOHzTAyonNcBg==",
    "payload" : "EHmAqzNGfYFH6xqkNLTP56UKFQMx8DotvDyzAFHbPHDYejIBbrqufh7Hg2F3G
    aDGbbnbgB7oAXXUdPHIDEgccpJEZ+6ewX184o7X4MdJBeZbS8bKcwdcoxCLt6fwa/V+YpLdu0o
    30SIFDKAv/hfL5A=="
  }
}
```

Listings

List of Figures

Glossary

2PC Two-Party Computation.

A-ABE Accountable Authority Attribute-Based Encryption.
ABAC Attribute-Based Access Control.
ABE Attribute-Based Encryption.
AES Advanced Encryption Standard.
API Application Programming Interface.

BN Barreto-Naehrig.

CL Client.
CLI Command Line Interface.
CP-ABE Ciphertext-Policy Attribute-Based Encryption.
CPU Central Processing Unit.
CSP Cloud Service Provider.

DH Diffie-Hellman.
DLP Discrete Logarithm Problem.
DO Data Owner.
DS Decryption Server.

EC Elliptic Curve.
ECDLP Elliptic Curve Discrete Logarithm Problem.

Fuzzy IBE Fuzzy Identity-Based Encryption.

GUI Graphical User Interface.

HTML Hypertext Markup Language.
HTTP Hypertext Transfer Protocol.

IAIK Institute of Applied Information Processing and Communications.

IBE Identity-Based Encryption.
IIoT Industrial Internet of Things.
IoT Internet of Things.

JSON JavaScript Object Notation.

KA Key Authority.
KP-ABE Key-Policy Attribute-Based Encryption.

LSSS Linear Secret Sharing Scheme.

mCP-ABE Mediated Ciphertext-Policy Attribute-Based Encryption.
MPC Multi-Party Computation.
MSP Monotone Span Program.

PBC Pairing-Based Cryptography.

RBAC Role-Based Access Control.
REST Representational State Transfer.
RREF Reduced Row-Echelon Form.
RS Re-Encryption Server.
RSA Rivest–Shamir–Adleman.

SHA Secure Hash Algorithm.
StS Storage Server.

TU Graz Graz University of Technology.

UUID Universally Unique Identifier.

Bibliography

[1] Gora Adj, Alfred J. Menezes, and Thomaz Oliveira. "Computing Discrete Logarithms in $\mathbb{F}_{3^{3\cdot 137}}$ and $\mathbb{F}_{3^{3\cdot 163}}$ Using Magma." In: *Arithmetic of Finite Fields*. Vol. 9061. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 3–22. ISBN: 978-3-319-16276-8. DOI: 10.1007/978-3-319-16277-5 (cit. on p. 7).

[2] Paulo S. L. M. Barreto and Michael Naehrig. "Pairing-Friendly Elliptic Curves of Prime Order." In: (2006), pp. 319–331. DOI: 10.1007/11693383_22 (cit. on p. 7).

[3] Paulo S. L. M. Barreto et al. "Efficient pairing computation on supersingular Abelian varieties." In: *Designs, Codes and Cryptography* 42.3 (2007), pp. 239–271. ISSN: 0925-1022. DOI: 10.1007/s10623-006-9033-6 (cit. on p. 7).

[4] *Base64*. URL: https://tools.ietf.org/html/rfc4648 (visited on 05/09/2019) (cit. on p. 44).

[5] Amos Beimel. "Secret Secure Schemes for Sharing and Key Distribution." PhD thesis. Israel Institute of Technology, 1996, p. 115 (cit. on p. 9).

[6] John Bethencourt, Amit Sahai, and Brent Waters. "Ciphertext-policy attribute-based encryption." In: *Proceedings - IEEE Symposium on Security and Privacy* (2007), pp. 321–334. ISSN: 10816011. DOI: 10.1109/SP.2007.11 (cit. on pp. 2, 9, 14, 16).

[7] Dan Boneh and Matthew Franklin. "Identity-Based Encryption from the Weil Pairing." In: *Advances in Cryptology — CRYPTO 2001* (2001), pp. 213–229. ISSN: 0097-5397. DOI: 10.1137/S0097539701398521 (cit. on pp. 13, 14).

[8] Melissa Chase and Sherman S.M. Chow. "Improving privacy and security in multi-authority attribute-based encryption." In: *Proceedings of the 16th ACM conference on Computer and communications security - CCS '09* (2009), p. 121. ISSN: 15437221. DOI: 10.1145/1653662.1653678 (cit. on p. 15).

[9] Sanjit Chatterjee and Alfred J. Menezes. "On cryptographic protocols employing asymmetric pairings the role of Ψ revisited." In: *Discrete Applied Mathematics* 159.13 (2011), pp. 1311–1322. ISSN: 0166218X. DOI: 10.1016/j.dam.2011.04.021 (cit. on p. 8).

[10] L Chen, Z Cheng, and Nigel P. Smart. "Identity-based key agreement protocols from pairings." In: *International Journal of Information Security* 6.4 (2007), pp. 213–241. ISSN: 1615-5262. DOI: 10.1007/s10207-006-0011-9 (cit. on p. 8).

[11] Ivan Damgård et al. "Multiparty Computation from Somewhat Homomorphic Encryption." In: *CRYPTO 2012*. Vol. 7417 LNCS. 2012, pp. 643–662. ISBN: 9783642320088. DOI: 10.1007/978-3-642-32009-5_38 (cit. on pp. 11, 39, 45).

[12] Ivan Damgård et al. "Practical Covertly Secure MPC for Dishonest Majority – Or: Breaking the SPDZ Limits." In: *IACR Cryptology ...* 2013, pp. 1–18. DOI: 10.1007/978-3-642-40203-6_1 (cit. on p. 11).

[13] Morris J. Dworkin et al. *Advanced encryption standard (AES)*. Tech. rep. Gaithersburg, MD: National Institute of Standards and Technology, 2001. DOI: 10.6028/NIST.FIPS.197 (cit. on p. 2).

[14] *ECCelerate*^{TM}. URL: https://jce.iaik.tugraz.at/sic/Products/Core_Crypto_Toolkits/ECCelerate (cit. on pp. 39, 44, 50, 59, 60).

[15] Taher ElGamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms." In: *Advances in Cryptology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 10–18. ISBN: 9783540156581. DOI: 10.1007/3-540-39568-7_2 (cit. on p. 19).

[16] Richard William Farebrother. *Linear Least Squares Computations*. New York, NY, USA: Marcel Dekker, Inc., 1988. ISBN: 0-82-477661-5 (cit. on p. 60).

[17] David Ferraiolo and Richard Kuhn. "Role-Based Access Control." In: *15th NIST-NCSC National Computer Security Conference*. 1992, pp. 554–563 (cit. on p. 14).

[18] Pierre-alain Fouque and Mehdi Tibouchi. "Indifferentiable Hashing to Barreto–Naehrig Curves." In: *Progress in Cryptology – LATINCRYPT 2012*. 2012, pp. 1–17. DOI: 10.1007/978-3-642-33481-8_1 (cit. on p. 39).

[19] *FRESCO*. URL: https://github.com/aicis/fresco (cit. on pp. 37, 39, 45, 58, 60).

[20] Gerhard Frey and Hans-Georg Ruck. "A Remark Concerning m-Divisibility and the Discrete Logarithm in the Divisor Class Group of Curves." In: *Mathematics of Computation* 62.206 (1994), p. 865. ISSN: 00255718. DOI: 10.2307/2153546 (cit. on p. 7).

[21] Steven D. Galbraith, Keith Harrison, and David Soldera. "Implementing the Tate Pairing." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2002, pp. 324–337. ISBN: 3540438637. DOI: 10.1007/3-540-45455-1_26 (cit. on p. 7).

[22] Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. "Pairings for cryptographers." In: *Discrete Applied Mathematics* 156.16 (2008), pp. 3113–3121. ISSN: 0166218X. DOI: 10.1016/j.dam.2007.12.010 (cit. on p. 8).

[23] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to play ANY mental game." In: *Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC '87*. January 1987. New York, New York, USA: ACM Press, 1987, pp. 218–229. ISBN: 0897912217. DOI: 10.1145/28395.28420. URL: http://portal.acm.org/citation.cfm?doid=28395.28420 (cit. on p. 11).

[24] Vipul Goyal et al. "Attribute-based encryption for fine-grained access control of encrypted data." In: *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06* (2006), p. 89. ISSN: 15437221. DOI: 10.1145/1180405.1180418 (cit. on pp. 2, 9, 14, 16).

[25] *Gradle Build Tool*. URL: https://gradle.org/ (cit. on p. 37).

[26] Robert Granger, Thorsten Kleinjung, and Jens Zumbrägel. "Breaking '128-bit Secure' Supersingular Binary Curves." In: *Advances in Cryptology – CRYPTO 2014*. Vol. 8617. 2014, pp. 126–145. DOI: 10.1007/978-3-662-44381-1_8 (cit. on p. 7).

[27] Florian Hess, Nigel P. Smart, and Frederik Vercauteren. "The Eta pairing revisited." In: *IEEE Transactions on Information Theory* 52.10 (2006), pp. 4595–4602. ISSN: 00189448. DOI: 10.1109/TIT.2006.881709 (cit. on p. 7).

[28] Vincent C. Hu et al. "Guide to Attribute Based Access Control (ABAC) Definition and Considerations." In: (2014). DOI: 10.6028/NIST.SP.800-162 (cit. on p. 2).

[29] Junbeom Hur. "Improving security and efficiency in attribute-based data sharing." In: *IEEE Transactions on Knowledge and Data Engineering* 25.10 (2013), pp. 2271–2282. ISSN: 10414347. DOI: 10.1109/TKDE.2011.78 (cit. on pp. 9, 15, 16).

[30] *IAIK-JCE*. URL: https://jce.iaik.tugraz.at/sic/Products/Core_Crypto_Toolkits/JCA_JCE (cit. on p. 39).

[31] Luan Ibraimi et al. "Mediated Ciphertext-Policy Attribute-Based Encryption and Its Application." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 5932 LNCS. 2009, pp. 309–323. ISBN: 3642108377. DOI: 10.1007/978-3-642-10838-9_23 (cit. on pp. 9, 15).

[32] *IIoT by Honeywell*. URL: https://www.honeywellprocess.com/en-US/online_campaigns/IIOT/Pages/index1.html (cit. on p. 2).

[33] *Jackson*. URL: https://github.com/FasterXML/jackson (cit. on p. 44).

[34] Antoine Joux. "A One Round Protocol for Tripartite Diffie–Hellman." In: *Algorithmic Number Theory*. Vol. 1838. 3. 2000, pp. 385–393. ISBN: 978-3-540-67695-9. DOI: 10.1007/10722028_23 (cit. on p. 5).

[35] Henning Kagermann, Wolfgang Wahlster, and Johannes Helbig. *Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0*. Tech. rep. 2013. URL: https://www.bmbf.de/files/Umsetzungsempfehlungen_Industrie4_0.pdf (cit. on p. 1).

[36] M Karchmer and Avi Wigderson. "On span programs." In: *[1993] Proceedings of the Eigth Annual Structure in Complexity Theory Conference*. IEEE Comput. Soc. Press, 1993, pp. 102–111. ISBN: 0-8186-4070-7. DOI: 10.1109/SCT.1993.336536 (cit. on p. 9).

[37] Allison Lewko and Brent Waters. "Decentralizing Attribute-Based Encryption." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 6632 LNCS. 2011, pp. 568–588. ISBN: 9783642204647. DOI: 10.1007/978-3-642-20465-4_31 (cit. on pp. 9, 15, 40, 50, 71).

[38] Guofeng Lin, Hanshu Hong, and Zhixin Sun. "A Collaborative Key Management Protocol in Ciphertext Policy Attribute-Based Encryption for Cloud Data Sharing." In: *IEEE Access* 5 (2017), pp. 9464–9475. ISSN: 21693536. DOI: 10.1109/ACCESS.2017.2707126 (cit. on pp. 3, 16, 17, 20, 21, 65).

[39] Zhen Liu and Zhenfu Cao. "On Efficiently Transferring the Linear Secret-Sharing Scheme Matrix in Ciphertext-Policy Attribute-Based Encryption." In: *IACR Cryptology ePrint Archive* 2010 (2010), p. 374. URL: http://eprint.iacr.org/2010/374 (cit. on pp. 9, 67).

[40] Alfred J. Menezes, Tatsuaki Okamoto, and Scott A. Vanstone. "Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field." In: *IEEE Transactions on Information Theory* 39.5 (1993), pp. 1639–1646. ISSN: 15579654. DOI: 10.1109/18.259647 (cit. on pp. 5, 7).

[41] Victor S. Miller. "Short Programs for functions on Curves." In: *Unpublished* (1986) (cit. on p. 6).

[42] Atsuko Miyaji, Masaki Nakabayashi, and Shunzou Takano. "New explicit conditions of elliptic curve traces for FR-reduction." In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E84-A.5 (2001), pp. 1234–1243. ISSN: 09168508. DOI: 10.1017/CBO9781107415324.004 (cit. on p. 7).

[43] *MongoDB*. URL: https://www.mongodb.com/ (cit. on pp. 37, 40, 45, 46).

[44] D. Page, Nigel P. Smart, and Frederik Vercauteren. "A comparison of MNT curves and supersingular curves." In: *Applicable Algebra in Engineering, Communications and Computing* 17.5 (2006), pp. 379–392. ISSN: 09381279. DOI: 10.1007/s00200-006-0017-6 (cit. on p. 7).

[45] *Predix*. URL: https://www.ge.com/digital/iiot-platform (cit. on p. 2).

[46] Sebastian Ramacher. "Bilinear pairings on elliptic curves." MSc thesis. Graz University of Technology, 2015 (cit. on pp. 39, 67).

[47] R L Rivest, A Shamir, and L Adleman. "A method for obtaining digital signatures and public-key cryptosystems." In: *Communications of the ACM* 21.2 (1978), pp. 120–126. ISSN: 00010782. DOI: 10.1145/359340.359342 (cit. on p. 2).

[48] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. "Security and privacy challenges in industrial internet of things." In: *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15*. New York, New York, USA: ACM Press, 2015, pp. 1–6. ISBN: 9781450335201. DOI: 10.1145/2744769.2747942 (cit. on p. 2).

[49] Amit Sahai and Brent Waters. "Fuzzy Identity-Based Encryption." In: *Advances in Cryptology – EUROCRYPT 2005*. 2005, pp. 457–473. ISBN: 978-3-540-32055-5. DOI: 10.1007/11426639_27 (cit. on p. 13).

[50] Farzad Samie, Lars Bauer, and Jörg Henkel. "IoT technologies for embedded computing." In: *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis - CODES '16*. New York, New York, USA: ACM Press, 2016, pp. 1–10. ISBN: 9781450344838. DOI: 10.1145/2968456.2974004 (cit. on p. 2).

[51] Hovav Shacham. "New Paradigms in Signature Schemes." PhD thesis. Stanford University, 2005 (cit. on p. 8).

[52] Adi Shamir. "Identity-Based Cryptosystems and Signature Schemes." In: *Proceedings of CRYPTO 84 on Advances in cryptology*. 1985, pp. 47–53. ISBN: 9783540156581. DOI: 10.1007/3-540-39568-7_5 (cit. on p. 13).

[53] F. Shrouf, J. Ordieres, and G. Miragliotta. "Smart factories in Industry 4.0: A review of the concept and of energy management approached in production based on the Internet of Things paradigm." In: *2014 IEEE International Conference on Industrial Engineering and Engineering Management*. IEEE, 2014, pp. 697–701. ISBN: 978-1-4799-6410-9. DOI: 10.1109/IEEM.2014.7058728 (cit. on p. 1).

[54] *Siemens MindSphere*. URL: https://siemens.mindsphere.io/en (cit. on p. 2).

[55] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. Vol. 106. Graduate Texts in Mathematics. New York, NY: Springer New York, 2009. ISBN: 978-0-387-09493-9. DOI: 10.1007/978-0-387-09494-6 (cit. on p. 6).

[56] *Spring Boot*. URL: https://spring.io/projects/spring-boot (cit. on pp. 40, 46).

[57] *Thymeleaf*. URL: https://www.thymeleaf.org/ (cit. on p. 40).

[58] Osmanbey Uzunkol and Mehmet Sabır Kiraz. "Still wrong use of pairings in cryptography." In: *Applied Mathematics and Computation* 333 (2018), pp. 467–479. ISSN: 00963003. DOI: 10.1016/j.amc.2018.03.062. arXiv: 1603.02826 (cit. on p. 8).

[59] Frederik Vercauteren. "Optimal Pairings." In: *IEEE Transactions on Information Theory* 56.1 (2010), pp. 455–461. ISSN: 0018-9448. DOI: 10.1109/TIT.2009.2034881 (cit. on p. 7).

[60] Yongtao Wang et al. "Mitigating key escrow in attribute-based encryption." In: *International Journal of Network Security* 17.1 (2015), pp. 94–102. ISSN: 18163548. DOI: 10.6633/IJNS.201501.17(1).13 (cit. on p. 16).

[61] Andrew Chi-Chih Yao. "How to generate and exchange secrets." In: *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. 1. IEEE, 1986, pp. 162–167. ISBN: 0-8186-0740-8. DOI: 10.1109/SFCS.1986.25 (cit. on p. 11).

[62] Andrew Chi-Chih Yao. "Protocols for secure computations." In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. IEEE, 1982, pp. 160–164. DOI: 10.1109/SFCS.1982.38 (cit. on p. 11).

[63] Xuanxia Yao, Zhi Chen, and Ye Tian. "A lightweight attribute-based encryption scheme for the Internet of Things." In: *Future Generation Computer Systems* 49 (2015), pp. 104–112. ISSN: 0167739X. DOI: 10.1016/j.future.2014.10.010 (cit. on p. 16).

[64] Keliang Zhou, Taigang Liu, and Lifeng Zhou. "Industry 4.0: Towards future industrial opportunities and challenges." In: *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2015* (2016), pp. 2147–2152. DOI: 10.1109/FSKD.2015.7382284 (cit. on p. 1).

[65] Dominik Ziegler, Josef Sabongui, and Gerald Palfinger. "Fine-Grained Access Control in Industrial Internet of Things: Evaluating Outsourced Attribute-Based Encryption." In: *Proceedings of the 34th International Conference on ICT Systems Security and Privacy Protection - IFIP SEC 2019*. 2019 (cit. on pp. 49, 51–58, 60, 65).