Department of Informatics

Technical University of Munich

Master's Thesis in Informatics

# Multi-Party End-to-End Encryption for the Inverse Transparency Toolchain

Jonas Hagg

Department of Informatics

Technical University of Munich

Master's Thesis in Informatics

# Multi-Party End-to-End Encryption for the Inverse Transparency Toolchain

# Mehrparteien-Ende-zu-Ende-Verschlüsselung für die Inverse-Transparenz-Toolchain

| | |
|---|---|
| Author: | Jonas Hagg |
| Submission Date: | December 15, 2022 |
| Supervisor: | Prof. Dr. Alexander Pretschner |
| Advisor: | Valentin Zieglmeier |

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching, December 15, 2022 Jonas Hagg

# Acknowledgments

I want to express my appreciation to the Chair of Software and Systems Engineering of my supervisor Prof. Dr. Alexander Pretschner, for the opportunity to work on this thesis. In particular, I want to thank my advisor Valentin Zielgmeier for his patience, all the helpful hints, and the detailed feedback I received.

# Abstract

The concept of *Inverse Transparency* aims to protect data sovereignty while enabling the sensible usage of data. The *transparency toolchain* supports this approach by providing a technical framework. It ensures that all data accesses are transparently logged. This thesis contributes to the toolchain by designing and implementing an end-to-end encrypted protocol that allows users to share and revoke access to their encrypted logs. A survey of potential encryption techniques shows that hybrid encryption can fulfill the identified requirements best. The designed protocol resists against surreptitious forwarding attacks, malicious data owners and curious servers. It is implemented as a cryptographic library available in Go, Python, and Typescript. It assumes that a secure PKI is available and relies on the "JSON Web Encryption and Signing" standard.

The existing toolchain was adapted to maintain encrypted logs. The performance evaluation shows that the encryption layer introduces an acceptable overhead in the toolchain. The encryption of a single log for up to 170 users does not introduce a human-noticeable delay. If 100 encrypted logs are fetched from the server, the processing time increases by 55*ms* to 298*ms*. This shows that the designed protocol can be used in practice to share encrypted logs within the *transparency toolchain*.

# Contents

# 1 Introduction

With the rise of the modern internet more and more systems become digital. This transformation goes hand in hand with the collection of unprecedented amounts of data leading to the age of big data. On the on hand, this enables great possibilities for data-driven technologies to improve living conditions in our complex and modern world. On the other hand, it also introduces the risks such as surveillance and the abuse of power. This observation leads to the fundamental question of the ongoing digital transformation: How can data be used to drive innovation while protecting privacy? [1, 2]

The novel concept of *Inverse Transparency* [1] is concerned with this question. It aims to provide true data sovereignty for individuals while allowing the sensible usage of data to support digital innovation. This is achieved by tracking each access to sensitive data. Whenever a data consumer requests, aggregates, or analyses data of individuals, a log is created. Data owners gain transparent insights into the usage of their data through those logs: Who accessed which of my data and why? Moreover, data owners can define policies that restrict access to their data for data consumers. Only explicitly allowed data accesses are valid and succeed. This enables true data sovereignty because any harmful data usage can be discovered and may be legally prosecuted.

Zieglmeier and Pretschner [3] proposed a technical framework supporting this concept. In their work, they implemented a proof-of-concept *transparency toolchain*. Figure 1.1 is taken from their publication and shows the major components of their toolchain. It consists of two actors: The data owner and the data consumer. The latter aims to access some sensitive data of the former. Each data access is tracked by a monitor component (*Monitor*), which sends the log to a dedicated server (*Safekeeper*). A front-end application (*Display*) makes all data accesses transparent to a user by visualizing the captured logs. It also allows data owners to define policies that restrict the access to certain data.

## 1.1 Problem statement

It is crucial to the concept of *Inverse Transparency* that all data accesses are tracked. However, the illegal processing and usage of data do not become legal simply because the data owner is transparently informed about it [1]. The harmful usage must be legally prosecuted. This motivates the idea that tracked data accesses can be shared in the system. If a data owner assumes that a given data access does not adhere to the data privacy regulations, the log can be forwarded to the legal department. Consequently, a data owner should be able to specify certain users in the system who additionally can access the respective log. A revoking mechanism could further provide the possibility to withdraw access to a log, e.g. because the legal department has finished processing or because the log was shared accidentally. This ensures that data owners always have full control over the logs concerning them.

An important concern is the confidentiality of the collected logs. The communication between the servers in the toolchain is usually secured and encrypted via TLS [4]. The

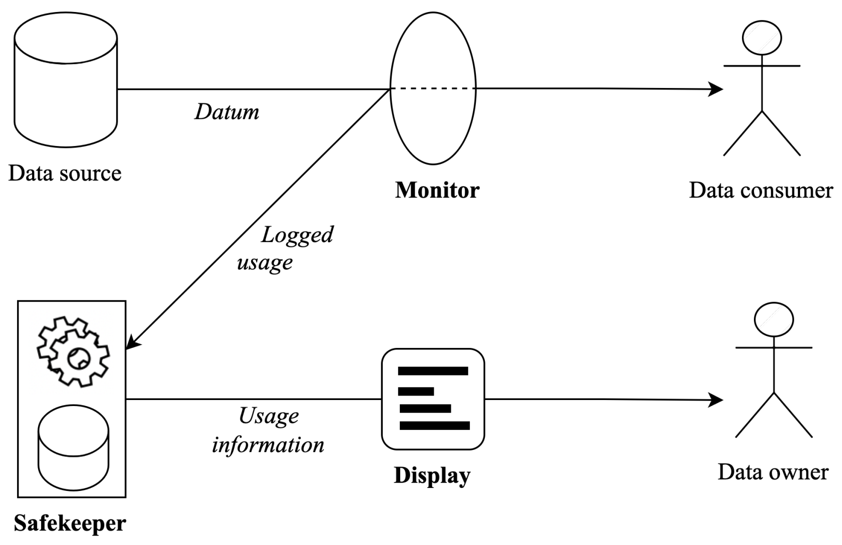


**Figure 1.1:** *The structure of the technical framework as proposed by Zieglmeier and Pretschner [3].*

processing servers (e.g. the *Safekeeper*), however, have full access to the logged data. This introduces several risks. First, the stored logs contain metadata about the tracked data access, e.g. the justification or the type of data that was accessed. Recent research shows that the analysis of metadata potentially undermines privacy [5, 6]. Since the log itself might contain sensitive information, it should be protected from unintended usage. Second, Zieglmeier and Pretschner [3] argue that their conceptual framework is instantiated by companies that have full access to the servers over which the toolchain is deployed. This generic approach enables great flexibility. Given the fact that most companies rely on cloud computing [7], however, this introduces the problem that the data might be stored on untrusted third-party servers. To avoid unintended access to the stored logs, the data passing those servers should be encrypted.

## 1.2 Contribution

This thesis contributes to the toolchain by designing and implementing an end-to-end encrypted protocol that allows data owners to share and revoke access to their logs.

Specifically, the thesis (1) designs a protocol based on a survey of multi-party encryption techniques, (2) implements the designed protocol and publishes it for the programming languages Typescript, Go, and Python, (3) integrates the implemented protocol into the existing toolchain, and (4) evaluates the implemented protocol in terms of functionality, security, and performance.

## 1.3 Outline

This initial chapter motivates the thesis. Chapter 2 covers the theoretical and technical background this thesis is based on. It includes fundamental cryptographic concepts and specifies the understanding of end-to-end encryption used throughout this work. The requirements of the intended protocol are investigated in Chapter 3. Functional and non-functional requirements are supplemented by security requirements, which introduce three attackers the protocol must defend against. Chapter 4 discusses different encryption techniques that can be used to implement those requirements. It also investigates modern end-to-end encrypted protocols and relates them to the concept proposed by this thesis. The theoretical design of the protocol is detailed in Chapter 5. It specifies how logs are signed, encrypted, and decrypted to fulfill the given requirements. Chapter 6 describes the cryptographic libraries implemented to realize the designed protocol. It also elaborates on the changes made in the existing toolchain to support encrypted logs. Given the requirements and the implementation, Chapter 7 evaluates the protocol in terms of functionality, security, and performance. Finally, Chapter 8 summarizes the major results of the thesis and concludes the thesis. It also discusses its limitations.

# 2 Background

This chapter covers the technical and theoretical background of the thesis. It provides short explanations of the concepts addressed in this work. Furthermore, it aims to clarify the understanding of the terms used throughout the thesis.

## 2.1 Cryptography

This section covers the cryptographic concepts the thesis relies on. Besides symmetric and asymmetric encryption, it details the theoretical motivation for hybrid encryption schemes. Modern end-to-end encrypted protocol demand authenticity of the encrypted data [8]. Thus, the concept of authenticated encryption is introduced.

### 2.1.1 Symmetric encryption

Symmetric encryption is characterized by the fact that the same key is used during encryption and decryption. This key is referred to as the symmetric encryption key. It must be transported securely from the sender to the receiver. Hence, symmetric encryption relies on a secure channel between the communicating entities. Symmetric encryption algorithms can be implemented very efficiently in hardware. For this reason, they are of great practical importance. [9, p. 300]

### 2.1.2 Asymmetric encryption

In contrast to symmetric encryption, asymmetric schemes do not use the same key for encryption and decryption. Each communicating partner is assigned a key pair consisting of a public key and a private key. While the private key must be kept secret, the public key must be known by other users and is thus publicly available. Asymmetric encryption schemes are characterized by the fact that data is encrypted using the public key of the receiver. The data can only be decrypted using the private key of the respective user. These schemes are built upon sophisticated mathematical constructions requiring complex computations. Hence, encrypting and decrypting big amounts of data via asymmetric schemes is not as efficient as in symmetric schemes. However, no secure channel is necessary between the communicating entities. [9, p. 331]

### 2.1.3 Hybrid encryption

Hybrid encryption schemes combine symmetric and asymmetric schemes to overcome their limitations. Symmetric schemes are used to encrypt the application data efficiently. The symmetric key is then encrypted by an asymmetric scheme for the intended recipients. Since this key is short, the overhead of the asymmetric encryption algorithm is acceptable. A recipient can restore the symmetric key because he can decrypt the encrypted key with his private key. Finally, he can decrypt the application data using the symmetric key.

This construction neither requires a secure channel nor suffers from the computational overhead of asymmetric schemes. [10]

### 2.1.4 Authenticated encryption

Classical symmetrical or asymmetrical encryption algorithms protect the confidentiality of data. However, integrity and authenticity are not guaranteed. Authenticated encryption overcomes this limitation. [9, 11]

Symmetric schemes can be improved by integrating integrity protection mechanisms, i.e. by hashing the content and attaching the hash to the cipher. If the integrity protection computation involves a secret, the data can also be authenticated. The resulting hash value is referred to as the authentication tag. In practice, those schemes are known as the AEAD encryption mode of block ciphers (e.g. AES-GCM). They ensure the integrity and authenticity of the symmetrically encrypted data. Users are authenticated if they know the symmetric key used to construct the cipher and the authentication tag. Note that if a symmetric encryption key is shared among a group of users, the authenticity is weakened. In this case, a decrypting party can only ensure that any of the users knowing the secret key created the cipher. [9, p. 315]

Asymmetric schemes provide stronger security guarantees. A digital signature proves that a message was signed by a certain user. If a digitally signed message is encrypted, the receiver can verify the originator of the message. Thus, combining public key encryption with digital signatures can also ensure the confidentiality, authenticity, and integrity of exchanged messages. [11]

## 2.2 JSON Object Signing and Encryption

"JSON Object Signing and Encryption" (JOSE) is a collection of standards defined by IETF [12]. It was created because traditional channel-based encryption layers such as IPsec [13] or TLS [14] only protect data between intermediate servers. However, those layers of encryption do not protect the data from the intermediate servers themselves. JOSE aims to provide security on the application layer, meaning that application data is embedded into secure objects. This protects the data from intermediate servers and enables the construction of end-to-end encrypted protocols. The JOSE specifications define three fundamental representations to serve this purpose [12]:

- The "JSON Web Signature" (JWS) format represents an integrity-protected object [15].
- The "JSON Web Encryption" (JWE) format represents an confidentiality-protected object [16].
- The "JSON Web Key" (JWK) format represents cryptographic keys [17].

Furthermore, a "JSON Web Token" (JWT) is a compact representation of JSON data [18]. It is passed as the payload to the construction of JWS and JWE tokens. A JWT token contains the JSON-encoded application data that must be protected. The protocol designed by this thesis relies on JWS and JWE tokens. The following explanations provide details about their internal structure.

### 2.2.1 JSON Web Signature

A JWS token contains integrity-protected data [15]. It supports both symmetric integrity protection and asymmetric integrity protection. Symmetric integrity protection can be realized using message authentication codes, while asymmetric integrity protection is often based on digital signatures. A JWS token consists of a header, a payload, and a signature. The payload contains the actual data that needs to be integrity-protected. The signature stores the MAC or the digital signature created by the sender of the message. This signature protects the payload and parts of the header. The header itself is split up into a protected header, which is integrity protected, and an unprotected header. As the name suggests, the signature does not protect the unprotected header. The header contains information about the algorithm used to secure the application data.

### 2.2.2 JSON Web Encryption

JWE tokens represent encrypted data [16]. Multiple encryption algorithms can be used to encrypt the application data. They are defined in the "JSON Web Algorithm" (JWA) specification [19]. JWE tokens enforce authenticated encryption and data can be encrypted for multiple recipients. Two algorithms are required to construct them: First, the application data is encrypted using an authenticated symmetric encryption algorithm. The key used for this algorithm is then encrypted with a dedicated key-wrapping algorithm. The receiver of the token must first apply the key-wrapping algorithm to restore the symmetric key, which then allows the decryption of the application data. A JWE token consists of the components listed in Table 2.1 [16]:

| **Field** | **Description** |
|---|---|
| Header | Contains information about the used algorithms. |
| Encrypted Key | Stores the encrypted key that was used to encrypt the application data. Multiple encrypted keys can be attached to the token. |
| Initialization Vector | Contains the initialization vectors of the encryption algorithms. |
| Additional Authenticated Data | Contains data that must be authenticated and integrity protected, but not encrypted. |
| Ciphertext | Stores the encrypted application data. |
| Authentication Tag | Stores the authentication tag that allows the receiver to validate the authenticity and integrity of a received token. |

**Table 2.1:** *Components of a JWE token.*

## 2.3 Key escrow

Key escrow refers to the scenario in which the cryptographic keys of users are known by a server in the system [20]. Some cryptographic schemes rely on a trusted key generation center that computes and distributes keys for all participants [21, 22]. This allows the center to access the private keys of all users meaning it can decrypt and sign arbitrary data. Key escrow can be beneficial in certain scenarios, e.g. to provide access to personal keys during vacation substitutions [23]. But it also conflicts with a strict interpretation of end-to-end encryption because it implies that not only the intended users can decrypt the data.

## 2.4 End-to-end encryption

The "National Institute of Standards and Technology" (NIST) defines end-to-end encryption (E2EE) as "[c]ommunications encryption in which data is encrypted when being passed through a network, but routing information remains visible" [24, p. 88]. Consider the scenario where Alice wants to send an end-to-end encrypted message to Bob. Once they established cryptographic keys, Alice encrypts the message and sends it to Bob. During transit, the encrypted message passes an untrusted network. In a classical understanding of E2EE, no third party or adversary within the untrusted network can access the message because it is encrypted [25]. If this holds, Alice and Bob are said to be the endpoints of the encryption and their communication is end-to-end encrypted.

Hale and Komlo pointed out, however, that this classical understanding of E2EE has changed in recent years [26]. With the rise of modern instant messaging services, E2EE should not only guarantee the confidentiality of the communication. Rather, an adversary should not have the possibility to influence communication in any way. This implies that modern E2EE-based systems should guarantee the authenticity and integrity of the exchanged data [26]. The difference between the classical and current understanding of E2EE can be described more precisely with the different adversary models they defend against.

### 2.4.1 Adversary models

This section details the security guarantees E2EE can provide by considering different adversaries. This differentiation shows that E2EE can be motivated by different attackers. It lays the foundation for the attacker models employed in this thesis because it clarifies what attacker's end-to-end encrypted systems can defend against.

The above NIST definition states that E2EE protects data passing an untrusted network. Within the network, the communicating entities do not have any control over their data. This motivates the idea that the attacker is interpreted as the whole untrusted network [27]. It is assumed that the adversary has limited computational capacity [28]. Specifically, it cannot break correctly implemented cryptographic schemes without knowing the key. In the following, active and passive attackers are further distinguished.

#### Passive adversaries

In general, passive attacks aim to break confidentiality through eavesdropping or unauthorized reading of data [9, p. 18]. In literature, this is often referred to as an honest-but-curious attacker [27]. The classical understanding of E2EE protects against passive adversaries. The attacker is part of the untrusted network and observes the passed data. It wants to access the content of the exchanged messages. To defend against this passive attacker, cryptography can be applied to make passive attacks ineffective and to protect confidentiality [9, p. 18].

#### Active adversaries

Today's interpretation of E2EE-based systems is motivated by a stronger attacker model [26]. An active attacker not only listens to the passing traffic. Rather, this attacker tries to insert, replace, drop, modify or replay messages to disrupt the communication [27]. More sophisticated active attacks also include spoofing or denial of service attacks [9, p. 19]. The motivation can be diverse: The attacker might want to access confidential content, introduce forged data, or delete valid data passing the network. Another goal might be the impersonation of a user participating in the system. Possible defense strategies against this adversary become more complex. The detection of inserted, replaced or modified data require some sort of validation that the received data was indeed created by the assumed communication partner (i.e. authenticity) and that it was not modified during transit (i.e. integrity). This can be realized with authenticated encryption techniques [8].

### 2.4.2 Endness

An intuitive understanding of E2EE implies that nobody - except the encryption endpoints - has access to the plaintext data. In an enterprise context, however, this might be not practical. The communication endpoints need to access the respective cryptographic keys to encrypt and decrypt data. Only the employee is allowed to know its cryptographic keys. If an attacker could compromise these keys, the communication is not protected anymore because he could simply decrypt observed traffic. If users lose their keys and the company does not have any backups, however, all their communication is lost. This can be mitigated by introducing a trusted server that stores the cryptographic keys of all employees. On the one hand, this can be practical because it increases the availability of data. On the other hand, such a trusted server introduces key escrow. The communication is not E2EE in a strict sense anymore because the administrator of the trusted server has access to all keys. Note that there is also the possibility to explicitly define this administrator as an additional encryption endpoint. In this case, there is no violation against E2EE because the administrator is assumed to be a valid communication partner. This example shows the fragility of E2EE: By defining arbitrary encryption endpoints, almost every system could claim to rely on E2EE. While this is technically true, it does not meet the common understanding of end-to-end encrypted systems [8, 26, 27].

# 3 Requirements

This chapter identifies the requirements of the protocol implemented in the context of this thesis. Overall, three major goals must be achieved:

I. Data owners can share and revoke access to their logs with other users in the system.
II. Log data must be confidential by means of end-to-end encryption.
III. Data owners cannot forge log data at any time.

Traditional requirements engineering methods usually differentiate between functional and non-functional requirements based on the views of different stakeholders [29]. Since this thesis operates within the security domain, the traditional requirements engineering process needs to be adapted: Precise assumptions about what to protect and against whom to protect are mandatory for a successful implementation and evaluation. Thus, this chapter closely follows the terms and methodologies proposed by Fabian et al. [30]. The authors fundamentally differentiate between functional, non-functional, and security requirements, which are extracted from the respective security goals. Goals are rather abstract and vague formulations of what the system should achieve. They are refined into more detailed requirements for two reasons [30]. First, verifiable requirements make it possible to compare different approaches. This leads to a reasonable choice from the solution space. Second, they allow the evaluation of the implemented solution.

While the overall goal I will be treated as a functional goal (see Section 3.2), goals II and III are understood as security goals (see Section 3.3). Further non-functional goals will be identified in Section 3.4. Within each subsection, the identified goals will be refined into requirements. This results in a list of verifiable and consistent system requirements for this thesis (see Section 3.1).

## 3.1 System requirements

This section summarizes all identified requirements elaborated in this chapter. They were derived from the major goals above. For details and the rationales of each requirement please have a look at the corresponding subsections below.

### Functional requirements

F1. Data owners can always access logs concerning them.
F2. Data owners can share the access to their logs with other users.
F3. Data owners can revoke the access to their logs from other users.

### Security requirements

S1. Logs can only be decrypted by authorized users.
S2. Logs cannot be forwarded to unintended receivers.
S3. Logs cannot be forged.

**Non-functional requirements**

N1. The protocol relies on a minimal number of trusted entities.
N2. The protocol utilizes standardized cryptographic algorithms.
N3. The protocol ensures the usability of the toolchain.
N4. The protocol utilizes a minimal amount of resources.

## 3.2 Functional requirements

Functional requirements describe "what the system does" [31, p. 11]. In the context of this thesis, they define the accessibility of logs in the *transparency toolchain*. Having access to a log effectively means that a user can decrypt the encrypted log. The following functional requirements were identified:

- Data owners can always access logs concerning them.
- Data owners can share the access to their logs with other users.
- Data owners can revoke the access to their logs from other users.

## 3.3 Security requirements

As proposed by Fabian et al. [30], this section determines the security goals of this thesis and refines them into security requirements.

Based on the major goals specified above and the identified functional requirements, the following security goals have been identified. They are motivated by malicious users that potentially participate in the process of sharing encrypted logs:

- Logs can only be decrypted by authorized users.
- Logs cannot be forwarded to unintended receivers.
- Logs cannot be forged.

These security goals can be refined into security requirements by annotating them with additional information, a counter-stakeholder, and specific circumstances [30]. Following this approach, each security requirement is specified by providing these details. The counter-stakeholder is an adversary attacking the system. Satisfying a security requirement should effectively defend against this adversary. The specified circumstances elaborate additional conditions that affect the security goal.

The following Section 3.3.1 clarifies the understanding of E2EE in this work. This specifies the security mechanisms that are required by the designed protocol. The subsequent sections refine the security goals into concrete security requirements. Those requirements are motivated by three attack vectors: First, a curious server extracts data from the stored log data. Second, a surreptitious forwarding attack forwards received logs without permission. Third, a malicious data owner inserts faked logs into the system. Each attack vector is detailed in the respective section.

### 3.3.1 E2EE in this work

This section details the understanding of E2EE in the context of this thesis based on the considerations in Section 2.4. Following the argumentation of Hale and Komlo [26], E2EE is understood as a defense against an active adversary. The attacker does not only observe the data but also tries to modify existing data or inserts forged data. Any defense against this attacker requires both, confidentiality and mutual authenticity.

**Confidentiality** The exchanged data needs to be encrypted. Otherwise, each untrusted entity can access the exchanged data and E2EE is broken.

**Receiver authenticity** The sender needs to be sure that only the intended receivers can decrypt the data. Specifically, no unauthorized entity must be able to decrypt the cipher. This requires special attention in the context of symmetric cryptography. Assume a group of users sharing a symmetric key. Data that is encrypted under this shared key can be decrypted by all members of the group. Thus, the set of encryption endpoints is equal to the group of users. When symmetric cryptography is used to construct E2EE-based protocols, messages between a single sender and a single receiver always require an exclusively exchanged symmetric key to ensure receiver authenticity. When operating with asymmetric cryptography, a key pair per receiver is required. If a receiver keeps the private key confidential, it can be ensured that only the receiver can decrypt data. This enables receiver authenticity.

**Sender authenticity** The receiver needs to be sure that the message was indeed sent by the assumed sender. Specifically, the receiver must ensure that the message was not created by an unknown entity. This requirement is crucial to defend against man-in-the-middle attackers. If a receiver cannot validate the origin of the message, an attacker could simply impersonate the sender and encrypt fake data under the public key of the receiver. Digitally signed messages ensure that a message was indeed sent by the claimed sender. Due to the risk of surreptitious forwarding, sender authenticity requires additional security mechanisms besides a plain digital signature [32].

### 3.3.2 Logs can only be decrypted by authorized users

In the context of the *transparency toolchain*, the intended E2EE should ensure that only authorized users can access logs. The passed log must be encrypted and only authorized users can decrypt it. Thus, this requirement demands confidentiality and receiver authenticity (details in Section 3.3.1). As specified by the functional requirements, the data owner can always decrypt the log. The data owner can also share the log with other users, which are then able to decrypt. Authorized users are the data owner of the log plus all users the data owner permitted access. From the perspective of E2EE, all other entities are untrusted. Untrusted entities must not be able to decrypt logs. The untrusted network includes the whole infrastructure of the toolchain, except the application a user interacts with. This implies that all front-end applications must be trusted, e.g. the *Display* component. The reliability of the created logs also depends on the integrated *Monitor* components. They must also be trusted.

The requirement is motivated by the risk of a curious server within the system [27]. Consider an attacker Eve who is the database admin of the *Safekeeper* server. Eve wants to know which monitor accesses which data frequently. Thus, he logs into the server and tries to extract this information from the database. If the stored logs are not encrypted, Eve can learn anything about the logs he might be interested in. This example demonstrates how a passive attacker could access the content of all logs stored in the toolchain. The attacker is passive because Eve does not participate in the protocol. He simply observes the stored data. The implemented protocol should defend against curious servers.

### 3.3.3 Logs cannot be forwarded to unintended receivers

The functional requirements demand that only the data owner of a log is allowed to share the log. No other user should be able to share the log in the name of the data owner. Whenever a user receives an encrypted log, it needs to be sure that the originator of the cipher is the data owner of the encrypted log. This requires sender authenticity (details in Section 3.3.1). Moreover, the receiver also needs to be sure that the data owner intended to share the log with him or her.

The following attack motivates this requirement. In literature, it is referred to as surreptitious forwarding attack [32]. Assume Alice to be a data owner and Eve to be a malicious user. Furthermore, assume that Alice shares a log with Eve. Eve has valid access to the log and can decrypt the encrypted log. Eve could apply the encryption algorithm and could specify Bob as the receiver. If the exchanged message does not contain information about the sender and the intended recipients, there is no way for Bob to know who actually sent the message and if the log was intentionally shared with him. Bob might assume that Alice sent the message because she is the data owner of the received log. Moreover, Bob could implicitly assume that he is a valid receiver because he can decrypt the message. In this scenario, however, Alice did not want to share the log with Bob. The log was simply forwarded by Eve without permission. This attack demonstrates how an active attacker could share logs with unintended users. The attacker forwards a valid log to an arbitrary user in the system. The implemented protocol must defend against surreptitious forwarding attacks.

### 3.3.4 Logs cannot be forged

A data owner can share logs with other users in the system. However, this introduces the risk that a malicious data owner shares forged logs with other users, e.g. by manipulating existing logs or creating arbitrary new logs. Only a log of a trusted and valid *Monitor* component should be accepted.

The following example motivates this requirement. Assume Eve to be a malicious data owner. Thus, he has access to all logs concerning him. Eve wants to harm a specific data consumer. To do so, Eve modifies an existing log or creates a new log indicating an illegal behavior of the targeted data consumer. Eve applies the encryption algorithm and shares the forged log with other users. Without further protection, the receiving users cannot verify the creator of the log. There is no way to detect this fraud. This example demonstrates how a malicious data owner could insert forged logs into the system. The implemented protocol must defend against malicious data owners.

## 3.4 Non-functional requirements

Non-functional requirements can be characterized as "global requirements on its development or operational costs, performance, reliability, maintainability, portability, robustness and the like" [31, p. 11]. In contrast to the functional and security requirements, the non-functional requirements are not a direct result of the overall requirements of the thesis. Thus, they are not strictly mandatory. At the same time, however, they add additional value to potential solutions. Consider two solutions *A* and *B*, which both satisfy all functional and security requirements. Solution *A* should be preferred over solution *B* if *A* resolves more non-functional requirements than *B*. During multiple iterations of elaborating potential solutions, the list of non-functional goals has grown to the following stable collection.

### 3.4.1 Minimal number of trusted entities

By definition, trusted entities are not part of the untrusted network. Thus, the proposed E2EE does not defend against them. This implies that a compromised trusted server undermines the security of the whole toolchain. To reduce the risk and attack surface, the toolchain should rely on a minimal number of trusted entities.

### 3.4.2 Standardized cryptographic algorithms

The currently implemented front-end of the *transparency toolchain* runs as a web application [3]. The Web Cryptographic API [33] defines an interface to perform cryptographic operations within browsers. It "exposes already existing and often heavily verified cryptographic functionality to Web application developers through a standardized interface" [34, p. 959]. This enables secure and performant cryptography within the browser [34]. Unfortunately, it is limited to specific primitives. Advanced encryption schemes, which are currently under research, are not defined and available in the API yet. To ensure the portability of the toolchain, the implementation of this thesis should thus only rely on primitives defined by the Web Cryptographic API. This is specifically necessary because the *Display* component is realized as a web application.

### 3.4.3 Usability

A trade-off between security and usability usually needs to be resolved during system design [35]. In the context of this thesis, however, the complex security is a key driver to improving usability. This is motivated by the principle of least effort, which states that humans will spend the least amount of work to get a task done [36]. If the user is asked to perform complex security tasks, he tends to neglect them. By integrating error-prone cryptographic tasks into the application, the usability can be improved because it simplifies the process for users. This also strengthens security because integrated mechanisms can be tested well. An optimal solution should integrate all cryptographic tasks into the toolchain. However, it comes with the cost of increased design and implementation efforts.

### 3.4.4 Minimal resource utilization

This is a performance criterion. The implemented solution should avoid resource overhead in terms of storage and computation. While this is directly related to the non-functional requirement N2, more aspects need to be considered here. Duplicated storage of data should be avoided. Specifically, an optimal solution does not store the same data encrypted with different keys in the database to avoid redundancy.

# 4 Potential solutions

This chapter theoretically elaborates on different approaches to implementing the requirements defined in Chapter 3. Each section introduces an encryption technique and discusses its consequences to the requirements of this thesis. The outcome of this chapter is a set of possible solution strategies. Moreover, the chapter shortly elaborates on modern protocols used in end-to-end encrypted applications. Finally, reasons are given why hybrid encryption is chosen as encryption technique within this thesis.

The following table summarizes the major results of this chapter. It provides an overview for the reader. A check (✓) indicates that an approach satisfies the corresponding requirement. A missing check means that the approach conflicts with the requirement. Details and justifications can be found in the corresponding sections below.

| Section | Approach | F1 | F2 | F3 | S1 | S2 | S3 | N1 | N2 | N3 | N4 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4.1 | External encryption | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ | | ✓ |
| 4.2 | Mutual encryption | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| 4.3 | Hybrid encryption | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ~ |
| 4.4 | Key server | ✓ | ✓ | ✓ | | ✓ | ✓ | | ✓ | ✓ | ✓ |
| 4.5 | ABE | ✓ | ✓ | ✓ | | ✓ | ✓ | | | ✓ | ✓ |
| 4.6.1 | IBBE | ✓ | ✓ | ✓ | | ✓ | ✓ | | | ✓ | ✓ |
| 4.6.1 | CBBE | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| 4.6.3 | PRE | ✓ | ✓ | ✓ | | ✓ | ✓ | | | ✓ | ✓ |

**Table 4.1:** *Overview of the considered encryption techniques and their implications to the system requirements.*

## 4.1 External encryption

An intuitive way of implementing a protocol that allows the exchange of encrypted logs among users is to rely on external tools. Suppose that Alice wants to share a log with Bob. Alice could send the log to Bob with an encrypted email. Emails can be encrypted with standardized protocols, e.g. PGP [37] or S/MIME [38]. They allow Alice to confidentially share logs with others. This approach, however, cannot satisfy the identified requirements. It is the responsibility of the user to correctly apply the encryption protocol. This is error-prone, it could lead to unintended security issues and it is not user-friendly. Sharing the log with $n$ users requires Alice to send $n$ encrypted emails. More importantly, this approach does not end-to-end encrypt logs because the *Safekeeper*, which stores the logs, still has access to the unencrypted data. Hence, the confidentiality of the logs is broken violating security requirement S1. Once an encrypted email is sent to the receiver, there is no way for Alice to revoke the access to the log. She loses control over the encrypted log because Bob's email provider stores the cipher. This is not in the control of the toolchain. Thus, Alice cannot revoke access to the log anymore. This breaks the functional requirement F3. Those limitations motivate the implementation of a more sophisticated protocol.

## 4.2 Mutual encryption

The term mutual encryption refers to the idea that logs are encrypted mutually for each authorized user. Each user in the system is assigned a key pair. This allows the data owner to encrypt a log for each user separately. The core idea is similar to the approach presented for external encryption (Section 4.1). However, this approach integrates the encryption and decryption of logs into the toolchain. This is detailed by the following example. Assume that a monitor component accesses data about Alice and creates a log. Since Alice needs to be able to access the log, the monitor encrypts the log under her public key and sends it to the server. If Alice keeps her private key secret, she is the only entity that can decrypt the log. She can download the log, decrypt it, and also share it with Bob by re-encrypting it under his public key. The re-encrypted data is then sent to the server where it is stored. Bob can finally download and decrypt the log.

All functional requirements are fulfilled: Alice can access the log because the monitor encrypted it with her public key. She can also share logs with others. The encrypted data is stored by the server. This allows Alice to request the deletion of the cipher from the server. Thus, Alice can revoke access to the log. As long as all users have exclusive access to their private keys, this approach also ensures the confidentiality of the logs. This fulfills the security requirement S1.

To defend against malicious data owners creating forged logs, the monitor component needs to cryptographically sign the log before encryption. If Alice shares the log, she does not share the plain log. Rather, she encrypts the signed log. Alice cannot modify the existing log or create a completely new log because she is not able to compute a valid signature in the name of the monitor. She can only succeed by knowing the private key of the monitor. This structure allows the receiver to verify that the log was created by a valid monitor component. As a consequence, a forged log can be detected during decryption, which adheres to security requirement S3.

Security requirement S2 can also be satisfied. To achieve this, Alice does not only encrypt the signed log directly. She additionally signs and includes the identity of the receiving user. This allows receivers to verify if the log was intentionally shared with them [32]. Figure 4.1 shows the structure of a *shared log* before it is encrypted. The actual log data is signed by the monitor. The *shared log* is a data structure that contains the signed log. It additionally specifies the user with whom the log is shared. The *shared log* itself is signed user creating it, i.e. the creator. The *shared log* is passed to the encryption algorithm. After successful decryption, a receiver receives this *shared log*. This allows him to validate if the sender intentionally shared the log. The receiver can also verify if the nested access log was signed by the claimed monitor.

This approach relies on standardized asymmetric cryptography. While it fulfills all functional and security requirements, it suffers from performance problems. Similar to external encryption, the whole log needs to be encrypted $n$ times if it is shared with $n$

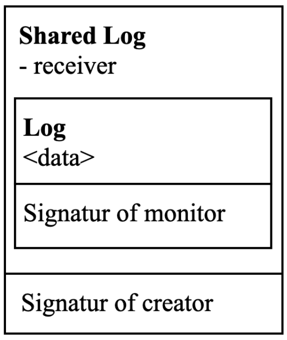


**Figure 4.1:** *Structure of a shared log before encryption.*

users. Much more problematic, however, is the application of asymmetric cryptography to application data. Public-key cryptography relies on intense mathematical computations and should not be used to encrypt large amounts of data directly [9, p. 340]. Eckert justifies this with the fact that asymmetric encryption algorithms are muss less performant than symmetric algorithms. Following her argumentation, asymmetric cryptography is usually used to encrypt a symmetric key, which finally encrypts the larger payload with a symmetric scheme. Thus, the implemented protocol should avoid the encryption of log data with asymmetric algorithms.

## 4.3 Hybrid encryption

This approach extends the concept introduced in the previous Section 4.2. Hybrid encryption combines the advantages of asymmetric cryptography with the advantages of symmetric cryptography. In a hybrid scheme, the content is encrypted via symmetric encryption. The short symmetric key is then encrypted with asymmetric algorithms. Thus, the expensive asymmetric algorithms only encrypt small amounts of data. It is used to securely transport the symmetric key. The larger payload is encrypted with an efficient symmetric scheme. [9, p. 340]

Consider the scenario where Alice wants to share data with Bob. Alice initially generates a symmetric key $k$. She uses $k$ to encrypt the payload with a symmetric scheme. She then encrypts $k$ under the public key of Bob. Finally, she sends the encrypted key along with the encrypted payload to Bob. To decrypt the payload, Bob first decrypts the symmetric key $k$ using his private key. In the final step, Bob can use $k$ to decrypt the symmetrically encrypted cipher.

This concept can be applied to the context of this thesis. An initially created log is encrypted for Alice using hybrid encryption. This results in a ciphertext that consists of the encrypted symmetric key and the encrypted log. The encrypted key can be decrypted only by Alice. Alice can use the key to decrypt the log. If she wants to share the log with Bob, she applies hybrid encryption again: This generates a fresh symmetric key and encrypts it under Bob's public key. Thus, the log is encrypted with the fresh key. The encrypted key and the encrypted log are sent to the server. This allows Bob to download and decrypt the log. Additionally, Alice can specify multiple receivers by attaching multiple encrypted keys to the encrypted log – one for each receiver. This is visualized in Figure 4.2. The key *k* is used to encrypt the log in a symmetric scheme. To make the key available to Bob and Charlie, the key is encrypted for both using asymmetric encryption. Alice can also revoke access to the log by re-encrypting it and updating it on the server.

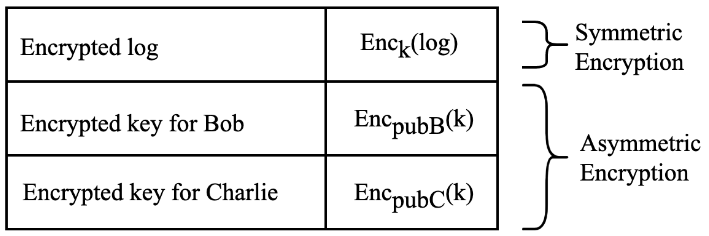

Encrypted log
$\mathrm{Enc}_k(\mathrm{log})$
Symmetric Encryption
Encrypted key for Bob
$\mathrm{Enc}_{\mathrm{pubB}}(k)$
Encrypted key for Charlie
$\mathrm{Enc}_{\mathrm{pubC}}(k)$
Asymmetric Encryption

**Figure 4.2:** *Structure of an encrypted log using hybrid encryption. The cipher contains the encrypted log and an encrypted key for each recipient.*

The above description shows that hybrid encryption techniques can be used to satisfy all functional requirements. Moreover, only those users can decrypt who were explicitly specified during encryption. This ensures that only authorized users can decrypt. This satisfies security requirement S1. To adhere to security requirements S2 and S3 the techniques introduced in Section 4.2 can be applied with a little modification. Instead of a single receiver, the *shared log* needs to contain a list of authorized users. The *shared log* is finally encrypted with the symmetric key. This allows receivers to validate if the log was intended for them and to verify if the actual log was modified during transit. Additionally, standardized algorithms exist for symmetric and asymmetric cryptography. This allows the construction of a secure and portable crypto library. While this approach improves the performance compared to Section 4.2, there is still a drawback. If the log is encrypted for *n* users, the symmetric key needs to be encrypted *n* times.

## 4.4 Key server

The development of cloud computing and centralized software architectures motivated the idea of key servers [39]. A key server stores cryptographic keys on behalf of the user. If a user authenticates against the server and has the required permissions, it can access a particular key. This can be useful in environments where multiple users need to access the same key. The creator and owner of a key can upload the key to the key server. If others require access to the key, the owner can modify the access policy.

The concept of a key server can be applied to share encrypted logs. Once a monitor creates a log because a data consumer accesses sensitive data of Alice, the monitor creates a symmetric key and encrypts the log under this symmetric key. The encrypted log is sent to the server. To decrypt the log, Alice requires access to this symmetric key. Thus, the monitor needs to upload the generated key to the key server, which allows Alice to download it. If she wants to share the log with Bob, there is no need to re-encrypt any data on the server. Rather, she needs to tell the key server that Bob is allowed to access this key. Alice can also revoke access to the file by telling the key server that a particular user is not allowed to access the key anymore. However, if a malicious user stored the key on his local machine, the revocation does not have any effect. Hence, the encrypted log in the server should be re-encrypted with a freshly generated key. This new key is then also uploaded to the key server.

This example shows that all functional requirements can be satisfied when utilizing a key server. It can be implemented using standardized symmetric encryption schemes (e.g. AES). The techniques introduced in Section 4.2 to satisfy security requirements S2 and S3 can be applied again. They ensure that logs cannot be forwarded and forged. Security requirement S1, however, cannot be met because the system suffers from key escrow: The administrator of the key server has access to all keys. The confidentiality of all logs is broken because this allows him to potentially decrypt all logs. The utilization of a key server also implies that we require an additional trusted component. This increases the attack surface of the system and does not adhere to the non-functional requirement N1.

## 4.5 Attribute-based encryption

Data access control mechanisms validate whether a user is authorized to perform an action on certain data [9, p. 242]. Access control is usually enforced by a server. Whenever a user requests access to data, the server checks if the user has the required permissions. A compromised server, however, can cause serious harm to a system because the server has unlimited access to all data stored on the server [23]. If data is not end-to-end encrypted, an attacker can access the data in plaintext. Attribute-based encryption (ABE) mitigates that threat. It shifts the logic of access control into the domain of cryptography [22]. Data is encrypted along with permissions. A user can only decrypt the data if it owns the demanded permissions.

Attribute-based encryption distinguishes between attributes and policies [22]. Attributes (e.g. $A, B, C$) are assigned to users. This assignment is encoded into a private key. Thus, the private key of the user contains attributes that are used to check if the user has certain permissions. Attributes can be used to create policies, e.g. $A \land B \land \neg C$. A policy is a logical expression of attributes. While encrypting data, the user specifies a policy that is encoded into the cipher. Successful decryption requires a private key that satisfies the encoded policy.

ABE can be used directly to encrypt data for multiple receivers [22]. During encryption, a policy is defined. All users that are equipped with attributes fulfilling this policy can decrypt data. By encoding the identities of valid receivers into the policy, the encrypting user could specify the set of users who can decrypt. If the access of a user needs to be revoked, the cipher is re-encrypted with a new set of receivers. This ensures that all required functional requirements are met. Again, one can equip the logs with cryptographic signatures (as described in Section 4.2). Those techniques fulfill the security requirements S2 and S3. This ensures that the logs cannot be forwarded and that logs can not be forged.

Attribute-based encryption usually relies on a trusted key generation center [40]. It computes and distributes cryptographic keys. This implies key escrow and harms security requirement S1 because the trusted server can potentially decrypt all ciphers. Additionally, each trusted entity increases the attack surface of the toolchain and conflicts with the non-functional requirement N1. Some schemes avoid a centralized trust center by employing a decentralized architecture [41]. These systems, however, cannot be integrated into the centralized architecture of the existing toolchain. A lot of papers are published in the field of attribute-based encryption. However, no cryptographic standard exists and it is not included in the Web Cryptographic API [33]. This conflicts with the non-functional requirement N2.

## 4.6 Broadcast encryption

Broadcast encryption (BE) is an encryption technique that allows to specify a set of authorized recipients during encryption [23]. Only those users can decrypt the produced ciphertext. It was initially proposed by Fiat and Naor [42]. In recent years, different realizations of broadcast encryption schemes were discussed [21, 23, 43, 44]. This section categorizes and summarizes those approaches as suggested in [23]. Furthermore, each approach is evaluated against the system requirements.

### 4.6.1 Identity-based broadcast encryption

Traditional identity-based encryption (IBE) schemes are public-key cryptography. The public key of a user is a human rememberable string, e.g. the email address. For a given public key the private key can be generated using a dedicated derivation function. Those private keys must be computed by a trusted entity that has access to secret system parameters. [45]

This basic idea was later adopted to implement broadcast encryption [23]. The resulting schemes are called identity-based broadcast encryption (IBBE) [21]. The following example illustrates its functionality [23]: Assume that Alice and Bob own unique email addresses. First of all, the IBBE scheme needs to be instantiated. The trust center computes the private keys on behalf of Alice and Bob, based on their email addresses. It also distributes the private keys and public system parameters to both users. Alice wants to send confidential data to Bob. She applies the encryption algorithm, which requires a set of public keys as input. Then she uploads the cipher to the server. This allows Bob to download and decrypt it using his private key. To revoke a user, Alice must re-encrypt the data and upload it again to the server.

The example shows that all three functional requirements can be fulfilled. The techniques from Section 4.2 can be applied again. This satisfies the security requirement S2 and S3, which ensure that logs can not be forwarded or forged. IBBE schemes suffer from key escrow because a trusted party computes and distributes private keys [23]. This breaks the confidentiality of the encrypted logs violating security requirement S1. The additional trusted entity also increases the attack surface, which breaks the non-functional requirement N1. A lot of research is going on in the field of IBE [23]. However, no cryptographic standard exists for IBE or IBBE. It is not included in the Web Cryptographic API [33]. This conflicts with the non-functional requirement N3.

### 4.6.2 Certificate-based broadcast encryption

Certificate-based encryption (CBE) was initially proposed in 2003 [20]. The following observations motivated this approach [23]:

- Traditional public key encryption schemes are faced with the certificate revocation problem [20].
- Identity-based encryption schemes usually rely on trusted entities. This introduces key escrow [23]. However, they do not suffer from the certificate revocation problem.
- Traditional public key schemes can be combined with identity-based schemes by double encrypting the plaintext. In particular, the plaintext is encrypted once under the public key of the traditional scheme and once under the public key of the identity-based scheme. The result of this construction is a CBE scheme. They eliminate key escrow and the certificate revocation problem. [23]

This idea was later adopted to broadcast encryption yielding a certificate-based broadcast encryption scheme (CBBE) [43]. Although this construction relies on a key generation center, it does not suffer from key escrow [23]. Once keys are established, a user can encrypt data for dedicated recipients. Only the users specified during encryption can decrypt the cipher. The key generation center cannot decrypt the cipher because it does not have access to any private key of the traditional scheme [23]. Besides the functional requirements, CBBE also satisfies the security requirements. The distinction between a shared log and a signed log introduced in Section 4.2 can be applied again. Thus, all receivers can validate that they are intended decryption endpoints and that the access log was created by the claimed monitor. This satisfies security requirements S2 and S3. The notion of E2EE is also fulfilled because the trusted server cannot decrypt the logs.

Unfortunately, no reference implementations of CBBE exist. This fact is a major drawback in the context of this thesis. These cryptographic systems are currently the subject of research. There are no ongoing standardization processes to analyze their security. As a result, the non-functional requirement N2 cannot be met. Using those schemes in a practical implementation is avoided in this thesis because the missing standards introduce an unpredictable risk.

### 4.6.3 Proxy re-encryption-based broadcast encryption

Proxy re-encryption (PRE) is another encryption technique proposed to realize broadcast encryption [23]. Its fundamental concept relies on a trusted proxy that transforms encrypted data between users. It is an important requirement of those schemes that the proxy does not learn anything about the plaintext during this re-encryption [46].

Similar to the example given in [23], consider a user Alice who stores a ciphertext $E_A$ in the cloud. Only Alice can decrypt $E_A$. Assume she wants to share the plaintext available with Bob. This requires Alice to share a re-encryption key $rk_{Alice,Bob}$ with the proxy. The knowledge of this key allows the proxy to transform $E_A$ to $E_B$. Bob can then download and decrypt $E_B$. Alice could revoke access to a log by instructing the server to delete the cipher for a dedicated user.

The proxy needs to be semi-trusted because with the knowledge of $rk_{Alice,Bob}$ it can re-encrypt all ciphers from Alice to Bob [46]. This has important consequences for the security of the protocol implemented in this thesis. Once Alice defined Bob as a valid receiver for a single log, the server could re-encrypt all ciphers of Alice for Bob. This, however, affects the confidentiality of encrypted logs. Not only authorized users might have access. Users might collude with the server, which re-encrypts logs on behalf of the user, to obtain decrypted logs. Hence, a proxy re-encryption-based scheme does not adhere to security requirement S1 because the confidentiality of logs can not be ensured. Notice that no cryptographic standard exists for proxy re-encryption-based cryptography. It is not included in the Web Cryptographic API [33]. This conflicts with the non-functional requirement N3.

## 4.7 E2EE in modern technology

Many modern applications rely on E2EE. This section exemplarily investigates the implemented protocols of Zoom, CloudSeal, and the messaging services Whatsapp and Signal. The goal of this section is to demonstrate which E2EE techniques are practically used in modern applications. Moreover, those approaches are discussed in the context of this thesis. All the following techniques have in common that they do not rely on a PKI. This restriction, however, does not hold in the context of this thesis because the *transparency toolchain* is intended to be realized in enterprises [3]. It is assumed that most of them have an established PKI, which simplifies the design of the final protocol.

### 4.7.1 Zoom

Zoom[1] is a video conference platform. It integrates E2EE to protect communication among participants. The enterprise claims that even the Zoom service provider cannot decrypt the exchanged data. [47]

Once a meeting is started, the initiator of the meeting generates a meeting key. This secret is shared among all users participating in the meeting. Thus, the key needs to be transported securely to all users joining the meeting. In particular, this requires the joining users to be online. All communication is encrypted using the meeting key with the authenticated symmetric encryption scheme AES-GCM. [48]

The protocol of Zoom is similar to the hybrid encryption described in Section 4.3. However, it cannot be applied in the context of this thesis because all participating users must be online. This assumption does not hold in the context of the *transparency toolchain*. Specifically, a monitor needs to be able to encrypt a log for the data owner even if the data owner is offline. The approach of Zoom shows, however, that hybrid encryption is practically used to send encrypted data to multiple receivers.

[1] `https://zoom.us/`

### 4.7.2 CloudSeal

CloudSeal [49] is a scheme to securely share and distribute files over cloud-based storages. Their E2EE relies on proxy re-encryption-based encryption (see Section 4.6.3 for details). CloudSeal can currently not be provided as a web application because it relies on advanced cryptographic algorithms, which are not included in the Web Cryptographic API. Thus, their clients are required to run on native operating systems. [49]

As highlighted in Section 4.6.3, proxy re-encryption-based schemes violate the identified requirements of this thesis. CloudSeal is exemplarily listed here because it shows that broadcast encryption schemes are practically implemented to achieve E2EE. It is not an option for this thesis because of the limited algorithms provided in the Web Cryptographic API.

### 4.7.3 Instant messaging services

Whatsapp[2] and Signal[3] are two popular instant messaging services. They claim to implement E2EE. Both services rely on the Signal protocol. The Signal protocol is a combination of the X3DH protocol [50] and the Double Ratchet protocol [51]. The X3DH protocol establishes a shared secret between two authenticated users. This effectively allows the initialization of a secure channel between the users. The shared secret is used in the Double Ratchet protocol to exchange encrypted messages. [50, 51]

The architecture of the Signal protocol establishes sessions between users. Encrypted data can be exchanged within those sessions. Encryption and decryption algorithms always require the initially established shared secret. The shared secret is a long-term key, which is usually stored in a secure hardware storage. When encrypting data for a set of users the plaintext is encrypted for each user separately. [52]

Unfortunately, the concept of sessions relying on an initially exchanged private key does not fit the architecture of the toolchain. The *Display* component allows users to log in to a web application. Within this application, the encrypted data is downloaded and decrypted. The sessions used by the Signal protocol require, however, that the exchanged shared key is available for the user during decryption. Since the user can log in from arbitrary devices and browsers, the long-term session key needs to be transported among those devices. Either the user handles this manually, which conflicts with usability, or a trusted server stores the session key, which conflicts with the notion of E2EE. In both cases, the requirements of this thesis cannot be met. This can be fixed by implementing the front-end as a mobile application. This way, a secure session could be established among users and the shared secret could be securely stored in dedicated secure hardware storages.

[2] `https://whatsapp.com/`
[3] `https://signal.org/`

## 4.8 Summary and evaluation

The protocol designed in this thesis relies on hybrid encryption. This section summarizes the findings in this chapter and justifies the usage of hybrid encryption. Recall Table 4.1, which summarizes the evaluation of all approaches. In particular, the table visualizes the fulfillment of the requirements for each considered approach. Hybrid encryption satisfies all functional and security requirements. Only the non-functional performance criterion N4 cannot be fully met because the symmetric key needs to be encrypted for all receivers. The following argumentation illustrates why hybrid encryption is nevertheless the most promising approach.

First of all, hybrid encryption improves mutual encryption because it avoids encrypting application data with asymmetric cryptography. Instead of encrypting a log multiple times under different public keys, hybrid encryption applies fast symmetric algorithms to encrypt the log. It then additionally encrypts the symmetric key under the public keys of the receivers. This adheres to currently proposed best practices in applied cryptography [9, p. 340]. The performance difference might not be noticeable in the current protocol because the encrypted logs do not contain a lot of information. However, the usage of hybrid encryption ensures that the encryption is still performant if the logs are extended in the future. Notice that there is no difference in terms of security and functionality between the approaches. Thus, hybrid encryption outperforms mutual encryption.

Another option to implement the intended protocol is based on broadcast encryption techniques. As elaborated in Section 4.6, different approaches are proposed to implement broadcast encryption. In general, there is no need to encrypt data $n$ times when it needs to be decrypted by $n$ users in those schemes. Hence, broadcast encryption schemes outperform hybrid encryption. The main disadvantage of broadcast encryption is the missing standardization of the algorithms. Specifically, the Web Cryptographic API does not include any of those schemes. Thus, they can only be used in the browser if the reference implementations are translated into Javascript. Since Javascript is an interpreted language, however, this comes with considerable performance limitations. Moreover, the cryptographic library implemented in this thesis needs to be delivered in three programming languages. Translating a dedicated scheme into multiple languages is complex, error-prone, and not realizable within the time constraint of this thesis. Since hybrid encryption relies on standardized and well-researched algorithms, it is preferred over broadcast encryption techniques.

A third option is the implementation based on a key server. The log is encrypted with a symmetric scheme and access to the decryption key is maintained by a key server. This approach only relies on well-researched symmetric encryption techniques. It can be implemented in different programming languages and environments. A key server, however, introduces key escrow harming the intended E2EE. This is a major drawback compared to hybrid encryption.

# 5 Design

This chapter describes and justifies the protocol designed in the context of this thesis. The main purpose of the protocol is to share encrypted logs with multiple parties. In the previous Chapter 4, possible encryption strategies were introduced and evaluated based on the identified system requirements (see Chapter 3). Hybrid encryption was identified as the most promising encryption technique. The evaluation of the protocol in terms of functionality, security, and performance can be found in Chapter 7.

A high-level overview of the designed protocol can be found in Section 5.1. Different design considerations for the protocol are discussed in Section 5.2. The imposed algorithms to create, encrypt, and decrypt logs are explained in the subsequent sections.

## 5.1 Protocol overview

The implemented protocol extends the current toolchain because it supports encrypted logs. Additionally, access to encrypted logs can be shared and revoked by the data owner.

The protocol relies on an established PKI. Each user requires two key pairs. The first key pair is dedicated to encrypting data. Its public key is called the encryption key while its private key is called the decryption key. The second key pair is intended to sign data. Its public key is called the verification key while its private key is called the signing key. Public keys are represented by PEM-encoded certificates signed by a trusted CA. Private keys are represented by a PEM-encoded private keys. This separation of keys is considered to be the best practice in modern key management [53, p. 33]. Each user must have exclusive access to its private keys. The public keys of all users are expected to be publicly available in the system. The protocol further assumes that a participating user has access to its key material, e.g. via hardware token.

The protocol relies on three fundamental cryptographic operations: Signing, encrypting, and decrypting logs. They are used to construct the full functionality of the demanded protocol. They enable the exchange of end-to-end encrypted logs. Moreover, they can be used to share and revoke access to logs. The following description delivers a high-level overview of the functionality of the protocol. This is additionally visualized in Figure 5.1.

Whenever a monitor component observes a data access, it must create a log. A log must always be a signed data structure. Thus, the protocol requires the monitor to perform two tasks. First, the raw log data needs to be cryptographically signed by the monitor. Second, the log needs to be encrypted for the data owner. The encrypted log is finally sent to the *Safekeeper*. This allows the data owner to request and decrypt the log. The decryption process also involves validity checks of the received data. This ensures that the protocol was applied correctly.

The data owner can share or revoke access to a log for other users. Therefore, the data owner needs to download and decrypt the log from the server. To share or revoke access to the log, the data owner applies the encryption algorithm with a set of desired users. The access can be shared by adding additional users during encryption. The access can be revoked by omitting users during encryption. It is crucial to the protocol that the data owner always specifies itself within the set of authorized users. Otherwise, the data owner cannot decrypt the log anymore. Finally, the user uploads the log to the server. Authorized users can download and decrypt the corresponding log. Again, validity checks of the received data ensure that the defined security requirements are respected.

Note that the encryption algorithm can be applied by the monitor and the owner of the log. Although only data owners are allowed to share and revoke access to logs, the log must be initially made available for the data owner. This can be done by encrypting the log for the data owner: Whenever a monitor created and signed a new log, it applies the encryption algorithm. The only recipient for a new log must be the data owner. To ensure that the monitor does not specify other unintended users, validation checks during decryption are mandatory.

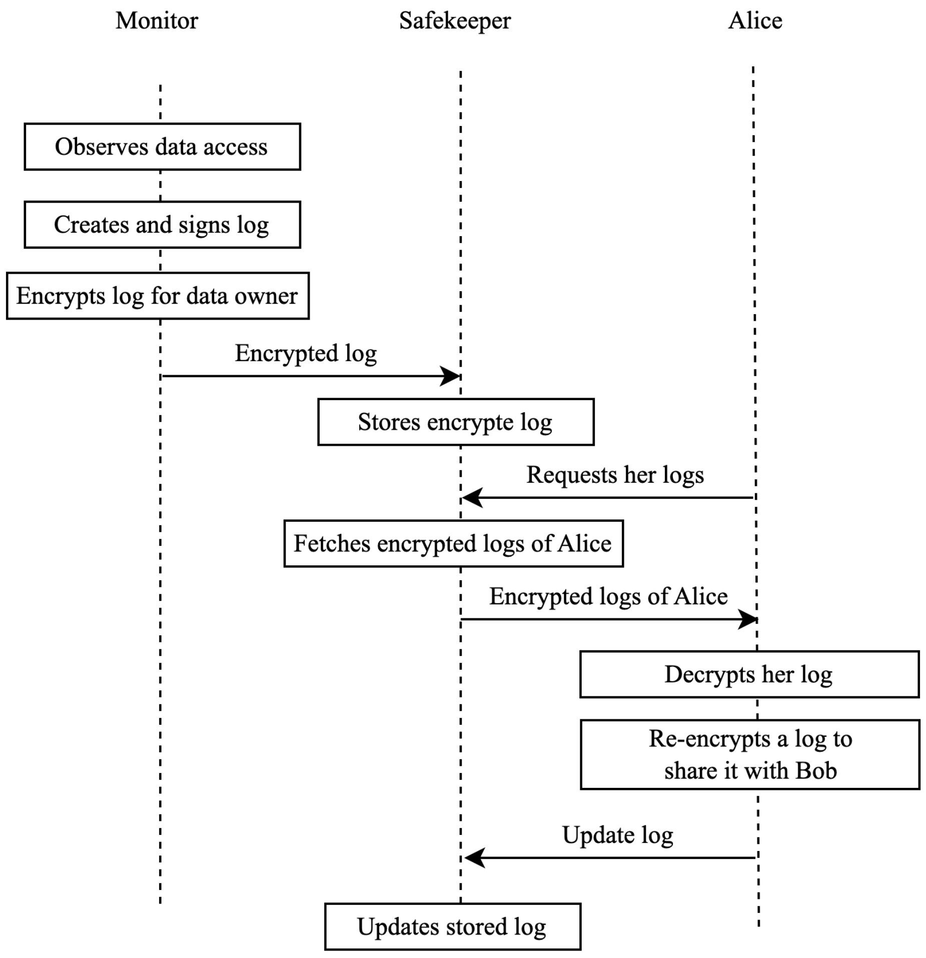


**Figure 5.1:** *The performed tasks and data flows when creating, and sharing logs in the toolchain. The monitor creates a log and encrypts it for Alice. Alice downloads and decrypts the log. Alice decides to share the log with Bob. Therefore, Alice re-encrypts the log and updates it in the* Safekeeper *component. Revoking Bob's access requires Alice to re-encrypt the log while omitting Bob from the set of recipients.*

## 5.2 General considerations

In this section, general design considerations of the protocol design are discussed. Section 5.2.1 evaluates the secure combination of encryption and authentication mechanisms. To design a practical and efficient protocol, intermediate servers are dependent on metadata. This is elaborated in Section 5.2.2. Finally, Section 5.2.3 illustrates how the JOSE standard can be utilized to practically implement the protocol.

### 5.2.1 Secure application of encryption and authentication

The protocol allows data owners to share encrypted logs with a set of recipients. The recipients must verify the authenticity of the encrypted data because only the owner or the monitor of a log are allowed to encrypt it. Thus, the creator of an encrypted log is required to digitally sign the sent data. This arises the question of whether encrypted data should be signed (*encrypt-then-sign*) or whether signed data should be encrypted (*sign-then-encrypt*). In general, the *encrypt-then-sign* approach is considered less secure because a malicious entity could simply stripe the signature and sign the ciphertext using its secret key [32]. This might lead to unintended flaws in the resulting protocol [18].

The naive *sign-then-encrypt* approach, however, suffers from surreptitious forwarding in certain scenarios [32]. The use case of this protocol is affected by the flaw. Consider a log that is signed by the data owner. This signed log is then encrypted for a set of users including a malicious user. Nothing prevents the malicious user from re-encrypting the received log for any other user in the system. It could finally forward the re-encrypted data to its target. The problem arises because the target cannot verify if the data owner has intended to share the log with him or her. They only know that the log was signed by the data owner. This flaw can be mitigated by explicitly signing the intended recipients. This allows a recipient to ensure that the data was intentionally encrypted for him by the claimed creator. [32]

For the protocol designed in the context of this thesis, those considerations have two implications. First, a log must be signed by the creator whenever it is shared. The creator must additionally sign the set of intended recipients. Note that the log itself is also a signed data structure. The log must always be signed by the monitor specified within the log to avoid the forgery of logs. This ensures the security requirement S3. The signature of the creator ensures that unauthorized sharing operations can be detected. This fulfills the security requirement S2. This construction nests singed data structures into each other. Details can be found in Section 5.4. Second, the signed data must be encrypted for the specified recipients. This produces a ciphertext that can only be decrypted by those recipients, which ensures the security requirement S1.

### 5.2.2 Metadata

The exchanged logs are always encrypted. This affects the performance of the toolchain. From the perspective of intermediate servers, an encrypted log does not provide any meaningful information. Specifically, they do not know which users have access to a log.

Hence, the server cannot associate a log with the authorized users. This introduces two practical problems.

The first problem occurs when Alice tries to request logs concerning her. If the server cannot associate logs with users, it must reply with all logs stored in the system. As a consequence, Alice receives many logs that she cannot decrypt. Since the raw cipher does not tell her which logs are intended for her, Alice must try to decrypt all logs to find the logs encrypted under her public key. This is very inefficient and requires a lot of computational power. To avoid this, the set of recipients can be attached as metadata to the encrypted log.

The second problem occurs when Alice tries to share access to her log. Consider Alice to be a data owner. Alice wants to share her log with Bob. Before the sharing process starts, the encrypted log is stored on the server. Alice re-encrypts the log under the public key of Bob. She authenticates against the server and tries to update the stored log. The server, however, does not know if Alice is the legitimate owner of the stored log because it cannot associate users with a given log. Thus, the server does not allow Alice to update her log. To avoid this, the server must know the identity of the data owner for all logs. This can be realized by attaching the identity of the owner as metadata to the encrypted log.

To mitigate these limitations, the set of receivers and the identity of the data owner must be accessible without decryption. This information can be interpreted as routing information and must be attached as metadata to the encrypted logs. It allows the server to filter logs concerning a particular user. Moreover, the server can allow data owners to modify their encrypted logs. This practically enables them to share and revoke access to logs. The construction implies, however, that the data sent over the insecure network contains the set of recipients twice. Once within the metadata and once within the encrypted data.

### 5.2.3 Realization via JOSE

A log within the current *transparency toolchain* is represented by a JSON object. This motivates the usage of the JOSE standard [12]. Details about it can be found in Section 2.2. The JOSE standard can be used to cryptographically sign (JWS) and encrypt (JWE) JSON data. It defines multiple algorithms that can be used. Specifically, JWE tokens can be computed using hybrid encryption. Since JWS and JWE tokens can also be nested into each other, JOSE can be used to realize the desired protocol by the following means:

- A log is a JWS token signed by the monitor. It is referred to as the inner JWS token because it is nested into another JWS token.
- To encrypt a log, the creator computes a JWS token, which contains the log as a nested token. The token additionally contains the set of intended recipients and the identity of the owner. This outer JWS token is called a *shared log* in the context of this protocol. See Figure 5.2 for a visualization of the nested tokens.
- The *shared log* is finally encrypted to obtain a JWE token. This token also contains the unencrypted metadata within its protected header. The metadata, however, is still integrity protected.

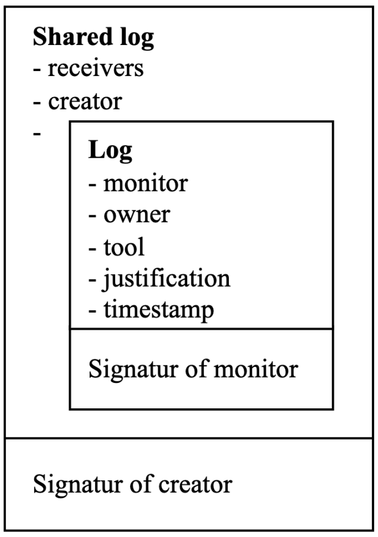


**Figure 5.2:** *A log is a JWS token signed by the monitor. A shared log is a JWS token that contains a nested log. It is signed by the entity encrypting the log.*

## 5.3 Creating logs

A log is represented by a JWS token. It contains the identity of the owner, the identity of the monitor, and further information that is relevant to identify the data access (see Figure 5.2). To avoid malicious data owners manipulating or creating new logs, each log needs to be signed by the monitor specified within the log. To create a JWS token, the monitor requires access to its private signing key. A log is signed with the algorithm ES256, which relies on ECDSA and SHA-256. It is defined by the "JSON Web Algorithm" specification [19]. The log is later used as input for the encryption algorithm. This construction is intended to resolve security requirement S3 because it allows decrypting users to verify if the provided log was indeed created by the claimed monitor. This resists the attack of a malicious data owner trying to forge logs.

## 5.4 Encrypting logs

The encryption algorithm requires two input parameters: A log and the set of users who are allowed to access the log. It can be performed either by a monitor or the data owner. A monitor initially encrypts the log for the data owner. The monitor must not include his own identity in the set of authorized users. More specifically, the set of authorized users must only contain the data owner if a monitor initially encrypts a log. The data owner of the log might decide to share or revoke access to a log for users later on. This requires the re-encryption of the log. The data owner must always include his own identity in the set of authorized users. The encryption algorithm outputs a JWE token, which can be decrypted by the specified users.

Internally, the encryption algorithm creates another JWS token. This token is signed with the private signing key of the user encrypting the log (i.e. the creator). It is called a *shared log* because it contains a nested log plus the set of recipients plus the identity of the creator (see Figure 5.2). The latter allows a decrypting user to download the public key of the creator, which is used to verify the signature of the *shared log*. The set of recipients is included to mitigate surreptitious forwarding. This construction is intended to resolve security requirement S2. It allows to verify whether the data owner has intended to share the log with a respective receiver. This resists surreptitious forwarding attacks [32].

Once the shared log is created, it is used to compute the final JWE token. Details about JWE tokens can be found in Section 2.2.2. This computation follows the *sign-then-encrypt* approach elaborated in Section 5.2.1. The final token is computed by passing the encoded shared log as plaintext into the JWE encryption algorithm. The identities of recipients and the identity of the owner are additionally passed as metadata into the protected header of the JWE token. The choice of used algorithms to encrypt the data implies hybrid encryption. The plaintext is encrypted with the authenticated encryption algorithm `A256GCM`. This is a symmetric encryption algorithm based on 256-bit AES using the Galois/Counter-Mode [19]. The used symmetric key must then be encrypted for each intended recipient. This allows the distribution of the key to all recipients. It is realized using the key-wrapping algorithm `ECDH-ES+A256KW`. The algorithm returns a cipher that can be decrypted by the receiver. It must be executed once for each receiver. The specification of `ECDH-ES+A256KW` requires the following steps to encrypt the symmetric key $k$ for a single receiver [54, p. 100]. This is also illustrated in Figure 5.3.

1. The sender must know the static public key of the receiver. In our case, this is the public encryption key. The receiver must have exclusive access to its static private key. In our case, this is the private decryption key.
2. The sender creates a new ephemeral key pair. This key pair must be used only once.
3. The sender sends the ephemeral public key to the receiver. In our case, the key is encoded into the JWE token. The ephemeral private key must be kept secret.
4. The sender knows the static public key of the receiver and the ephemeral private key of the sender. The receiver knows the ephemeral public key of the sender and the static private key of the receiver. Thus, the Diffie-Hellman key agreement protocol can be used to compute a shared secret between the sender and the receiver [9, p. 438]. The computation of the shared secret requires either the ephemeral private key or the static private key. If both are kept secret, it is assumed to be computationally infeasible to compute the shared secret [9, p. 438].
5. The shared secret, which is the result of the Diffie-Hellman key agreement protocol, is used as input for a key derivation function. This function computes an ephemeral key between the sender and the receiver.
6. Finally, this ephemeral key is used as key to encrypt the symmetric key $k$ for the receiver. This encryption uses the symmetric scheme AES.

The resulting JWE token contains the symmetrically encrypted log, the encrypted keys for all recipients, and the protected header. It can only be decrypted by the specified

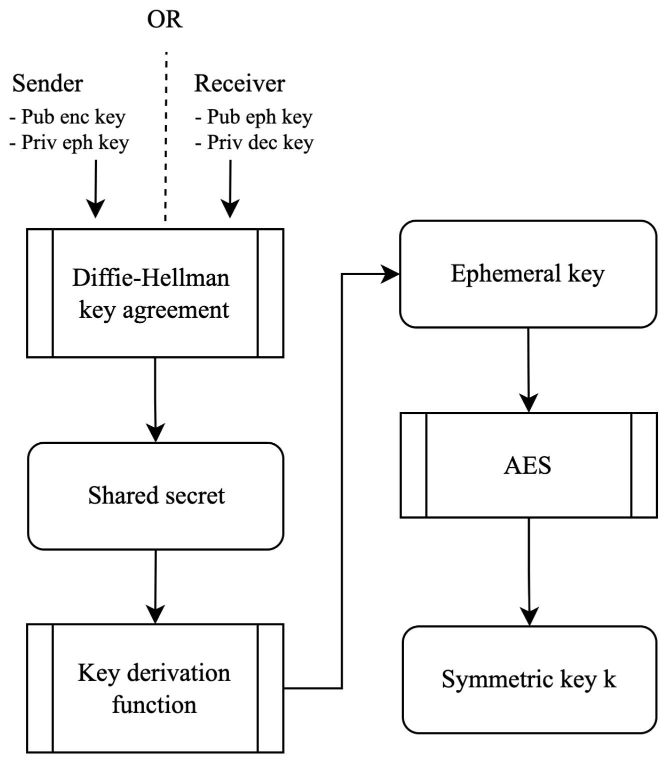


**Figure 5.3:** *Key-wrapping of the symmetric key k using ECDH-ES+A256KW. Based on the exchanged public keys the sender and receiver can both compute the same shared secret. The derived ephemeral key is finally used to encrypt/decrypt the symmetric encryption key k under AES.*

recipients because they are the only users who can restore the symmetric encryption key. This is intended to resolve security requirement S1 because it effectively end-to-end encrypts a log. Only explicitly authorized users can decrypt the log. The whole encryption process is depicted in detail in Appendix A.1.

## 5.5 Decrypting logs

The decryption algorithm takes a JWE token and a private decryption key as input. To verify the JWS tokens encoded within the JWE token, this algorithm needs to be able to dynamically resolve the identities of users to their public keys.

First of all, the JWE token is parsed into its ciphertext, header, and encrypted keys. The JWE decryption algorithm is then applied to the cipher. This decrypts the ciphertext using the passed private decryption key. First, it tries to decrypt any of the encrypted keys provided in the JWE token using `ECDH-ES+A256KW`. If this succeeds, the symmetric encryption key is accessible. This allows the decryption of the symmetric ciphertext using `A256GCM`. This effectively restores the *shared log* for the receiver. No decryption is necessary to access the metadata because is stored in the protected header of the JWE.

If the decryption is successful, the decrypting user has access to the *shared log* and the metadata. Multiple validations are necessary to ensure that the protocol is used correctly. If any of them fails, the decryption must be aborted with an error.

1. The signature of the obtained *shared log* (outer JWS token) must be validated. To do so, the decrypting user needs to download the public key of the creator specified in the *shared log*. If the verification of the signature succeeds, the decrypting user can be sure that the claimed creator indeed signed the *shared log*.
2. The signature of the nested log (inner JWS token) must be validated. Again, the public key of the specified monitor needs to be downloaded. The PKI must ensure that this public key belongs to a valid and trusted monitor component. If the verification of the signature succeeds, the decrypting user can be sure that the claimed monitor indeed signed the log.
3. The metadata includes the set of recipients. The *shared log* also includes this information. Only if both sets are equal, the token was not modified during transit.
4. The user applying the decryption must be part of the set of recipients. If this is true, the user can ensure that the creator intended to share the log with him or her.
5. The owner specified in the metadata must be equal to the owner specified in the log. Only if both are equal, the token was not modified during transit.
6. The log specifies the owner and the monitor of the log. The *shared log* specifies the creator of the encrypted message. The creator must be either the owner or the monitor of the log.
   If the creator is equal to the monitor of the log, the monitor encrypted the message for the data owner. In this case, the decrypting user must be the data owner. The set of recipients must only contain the identity of the data owner because the monitor is not allowed to encrypt the log for other users.
   On the other hand, the creator can be equal to the owner of the log. This implies that the owner shared the log with other users. In this case, the set of recipients can be arbitrary.

If all those validations succeed, the received log can be trusted. A visualization of the decryption process can be found in Appendix A.2

# 6 Implementation

This chapter describes the libraries that were developed in the scope of this thesis. They are available in the programming languages Python, Go, and Typescript. This ensures that software components in different environments can handle logs as specified by the protocol. Details about this practical realization of the libraries can be found in Section 6.1. Moreover, the chosen approach heavily influences the existing toolchain. The existing source code of the toolchain was refactored and adjusted to handle encrypted logs. This is described in Section 6.2.

## 6.1 Implemented Libraries

The following table gives an overview of the implemented libraries. It includes links to the GitHub repositories and the package indexes of the programming languages where the libraries were published. The acronym *it-crypto* stands for *Inverse Transparency Cryptography*.

| Name | Language | GitHub | Link to package index |
|---|---|---|---|
| Go-it-crypto | Go | haggj/go-it-crypto | PKG |
| Py-it-crypto | Python | haggj/py-it-crypto | PyPI |
| Ts-it-crypto | Typescript | haggj/ts-it-crypto | NPM |

**Table 6.1:** *Overview of the implemented cryptographic libraries.*

All three libraries are fully tested and compatible with each other. Data that was encrypted in one library can be decrypted by all libraries. It was taken care that classes, functions, constants, and variables are named similarly. Identical names are not possible because of the different naming conventions in Go, Python, and Typescript. Moreover, all libraries employ the same folder structure. This aims to improve the maintainability of the source code. As a result, all three libraries expose the same interface to users. This interface is highly related to the algorithms described in Chapter 5. The designed protocol relies on the JOSE standard. Thus, each library depends on a third-party package implementing JOSE. Go-it-crypto depends on the Go package *"JOSE"*[1]. Py-it-crypto depends on the Python package *"jwcrypto"*[2]. Ts-it-crypto depends on the Typescript package *"JOSE"*[3].

### 6.1.1 General usage

This section describes the usage of the implemented libraries in general. Please also refer to the pseudocode depicted in Listing 6.1. If you are interested in the concrete realization in a specific language, have a look at the corresponding GitHub repositories.

[1]`https://pkg.go.dev/gopkg.in/square/go-jose.v2`
[2]`https://pypi.org/project/jwcrypto/`
[3]`https://npmjs.com/package/jose`

To make use of the library an instance of type `ItCrypto` must be instantiated. This instantiation requires the function `fetchUser`. It resolves an identity of a user, which is a `string`, to an instance of type `RemoteUser`. The function usually requests a server to resolve the identity of a user to its public keys. It needs to implement the following signature: `RemoteUser fetchUser(string)`. A `RemoteUser` object stores the public keys of the user along with its identity and the information if the user is a trusted monitor. This information is mainly used during decryption to verify if the signed data is valid. A `RemoteUser` object should never be constructed directly. Rather, the `importRemoteUser` function must be called. This function expects a trusted certificate as a parameter. This trusted certificate is assumed to contain the public key of a root certificate authority within the PKI. The public keys of all users are assumed to be signed by this certificate authority. The function `importRemoteUser` verifies if the provided public keys are signed accordingly.

Once the `ItCrypto` object is instantiated, a user needs to log in. A logged-in user is represented by an `AuthenticatedUser` object. It extends the `RemoteUser` object because it additionally provides the private keys of the user. This allows the logged-in user to sign and decrypt data. A login is realized by calling the `login` method on the `ItCrypto` object. This function expects the PEM-encoded key material. The `ItCrypto` object becomes fully functional if a user is logged-in. This allows the user to create, encrypt, and decrypt logs using the provided keys.

### 6.1.2 Documentation ItCrypto

This section further details the `ItCrypto` object. It aims to provide a documentation for developers who integrate the library into their software. In particular, the methods `signLog`, `encryptLog`, and `decryptLog` are detailed. As described in Chapter 5, those methods can be used to provide the full functionality of the intended protocol.

#### Signing logs

The method `signLog` cryptographically signs the given log data. Monitors must call this encryption function to initially sign an instantiated log. The protocol does not allow any raw logs in the system. Rather, all logs must be signed by the monitor that created the log. This method relies on the private signing key of the logged-in user.

| Input parameters | |
|---|---|
| `accessData` | This is a data structure containing the raw log data. In particular, this raw data must be encodable as JSON data. It contains all relevant information to log the observed data access. |
| Output | |
| The method returns a JWS token encoded as string. It is signed by the logged-in user and it contains the JSON-encoded accessLog as payload. | |

**Table 6.2:** *Input parameters and output of the signLog method.*

```
RemoteUser fetchUser(string identity){
    /*
    Fetch keys from the server and check if a user
    is a trusted monitor.
    */
    return importRemoteUser(
        identity,
        /* keys */,
        isMonitor,
        rootCA
    )
}

// Initialize itCrypto object
var itCrypto = ItCrypto(fetchUser)
itCrypto.login(/* keys */)

// Sign log data
var signedLog = itCrypto.signLog(/* log data */)

// Encrypt log for a set of recipients
var alice = fetchUser("identity@alice.com")
var bob = fetchUser("identity@bob.com")
var loggedInUser = itCrypto.user
var encryptedLog = itCrypto.encryptLog(
    signedLog,
    [alice, bob, loggedInUser]
)

// Decrypt log
var decryptedLog = itCrypto.decryptLog(encryptedLog)
assert(decryptedLog == signedLog)
```

**Listing 6.1:** *Pseudocode of creating, encrypting, and decrypting logs using the provided libraries.*

### Encrypting logs

The method `encryptLog` encrypts a given log for a given set of recipients. Monitors must call this function to initially encrypt the log for the data owner. It is crucial to the security of the protocol that the monitor only chooses the data owner as a recipient. This encryption function can be used to share or revoke access to the log. In this case, it must be called by the data owner of the particular log. This method relies on the private signing key of the logged-in user.

| Input parameters | |
|---|---|
| `log` | This is the log that needs to be encrypted. It must be signed by a monitor. The output of `signLog` can be directly used here. |
| `recipients` | This is a list of `RemoteUser` objects containing the identity and public encryption key for each recipient. This set defines which users can decrypt the created cipher. |
| Output | |
| The method returns a JWE token encoded as string. It contains the provided signed log and can only be decrypted by the specified recipients. The token specifies the logged-in user as creator. It also contains the set of recipients and the identity of the data owner as metadata. | |

**Table 6.3:** *Input parameters and output of the encryptLog method.*

### Decrypting logs

The method `decryptLog` decrypts a given JWE token. A user can only decrypt this log if it was included as a recipient during encryption. It might be called by any user that received an encrypted log. The method performs multiple validations to detect any misusage of the protocol. In particular, it verifies all signatures of the encoded JWS tokens. This function relies on the private decryption key of the logged-in user.

| Input parameters | |
|---|---|
| `jwe` | This is a JWE token that represents an encrypted log. This token is the result of `encryptLog`. |
| Output | |
| The method returns a log. This is the same object that was passed to the encryption function. Therefore, it is still signed by a monitor. The signature, however, was already verified during decryption. This construction allows a data owner to easily re-encrypt the log. | |

**Table 6.4:** *Input parameters and output of the decryptLog method.*

### 6.1.3 Cryptographic keys

The libraries require cryptographic keys to sign, encrypt, and decrypt logs. As described in Section 5.1, the protocol assumes an externally established PKI. Each user is assigned two key pairs: One for signing and one for encrypting data. The private keys are assumed to be PEM-encoded. Public keys are assumed to be PEM-encoded certificates that are signed by a trusted certificate authority. Whenever a user is imported, the certificates of the user are assumed to be signed by this certificate authority. Currently, the libraries only support elliptic curve cryptography. The elliptic curve OpenSSL parameters `prime256v1` are known to work in all three implementations.

## 6.2 Toolchain modifications

The integration of the protocol requires an adaption of the existing toolchain. Figure 6.1 was taken from Zieglmeier and Pretschner [3]. It additionally visualizes the end-to-end-encrypted data flow in the toolchain. In particular, a monitor encrypts the log for the data owner. A user that is logged-in to the display component can decrypt its logs. Moreover, access to the logs can be shared and revoked. This functionality requires changes in the *Monitor*, *Safekeeper*, and *Display* components. Those changes were implemented in the context of this thesis. The development environment of the toolchain is a proof-of-concept. It shows how a PKI infrastructure can be used to integrate encrypted logs. The following sections describe different perspectives on this integration. They provide details about the implemented changes.

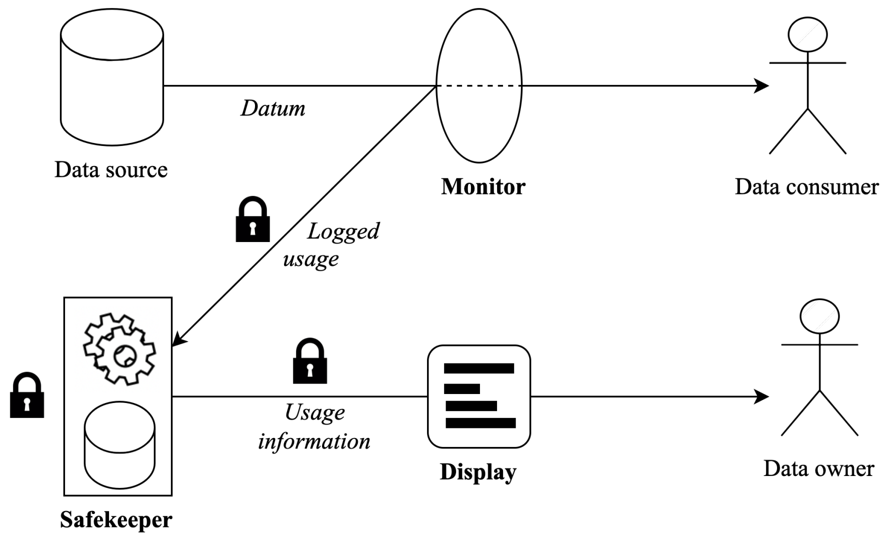


**Figure 6.1:** *The general data flow in the toolchain has not changed. The passed data, however, is end-to-end encrypted.*

### 6.2.1 User management

The toolchain relies on a single-sign-on server responsible for authenticating users. The development environment provides the *Revolori* server, which is an implementation of this service. Besides the general information of a user, it now additionally stores certificates containing the public keys of the users and the information if a user is an authorized monitor. This additional information is used by the front-end to encrypt and decrypt logs. Therefore, a new endpoint was introduced that resolves the identity of a user to this data.

The development environment of the toolchain is a proof-of-concept that shows how an external PKI can be connected. There are two scripts used to load pre-configured users into the development environment. Both were updated to establish an exemplary PKI.

The script `prebuild.sh` is responsible for creating a key pair for the *Revolori* server. It is used to sign access tokens. The public key is passed to the *Overseer* because it needs to verify if a provided token was signed by the *Revolori* single-sign-on-server. The script was extended to create a key pair for an exemplary certificate authority. Moreover, the script expects a file `users.json` that contains the pre-configured user data. Two key pairs are generated for each user. One is used to encrypt data and the other is used to digitally sign data. The public keys of all users are signed by the public key of the certificate authority. This effectively creates certificates binding the identities of the users to their public keys. Finally, the script updates the provided `users.json` to contain the generated cryptographic keys of each user.

The script `setup.sh` is responsible for creating users in a running *Revolori* instance. It is also used to create exemplary data accesses for each user. Thus, the script had to be adapted to support encrypted logs. First of all, the script expects the file `users.json` containing user information along with their cryptographic keys. It then parses the user information and creates the respective users in the *Revolori* server. In particular, this includes the certificates of the user's public keys. To pre-compute encrypted logs, the command-line interface of the *ts-it-crypto* library needs to be installed. It allows to encrypt log data in the name of a user. The encrypted logs are then uploaded to the *Overseer*.

This setup allows to fetch the public keys for a given user from the *Revolori* server. The development environment also provides the private keys for a user upon login. It is realized by passing the `users.json` file to the *Clotilde* process. This implies that the front-end has unrestricted access to all private keys. Thus, this setup can only be used in the development environment. In productive systems, the *Clotilde* front-end must securely import the private keys from the user, e.g. via hardware token. Whenever a user logs in to the system, the front-end instantiates an `ItCrypto` object using the provided key material. This finally enables all cryptographic tasks for a user.

### 6.2.2 Display component

The *Display* component is the front-end of the toolchain. It is instantiated as the *Clotilde* server in the development environment. It visualizes all data accesses concerning the logged-in user. Since all logs are end-to-end encrypted, it must decrypt the logs provided by the *Safekeeper*. Access to the logs can also be shared and revoked by the data owner.

This allows a data owner to authorize any other user in the system to temporarily access a log. The adapted implementation provides this functionality. To perform the necessary cryptographic actions, it integrates the *ts-it-crypto* library into the front-end application. Appendix B contains screenshots of the realized UI.

### 6.2.3 Safekeeper component

The *Safekeeper* component is responsible to store logs. It was modified to handle encrypted data. As described in Section 5.2.2, an encrypted log contains metadata allowing intermediate servers to distribute logs efficiently. This metadata contains the set of recipients and the identity of the data owner. Once a log is sent to the server, thus, this metadata must be parsed. It is stored along with the encrypted log in the database. All encrypted logs are stored in a single database table . It consists of three columns. The unique `id` is the primary key of the table. Additionally, two SQLite JSON fields[4] are used to store the log data, which contains the JWE token, and the metadata, which contains the parsed headers. Database queries can be run efficiently against those JSON fields to filter logs [55]. This allows the *Safekeeper* to associate a log with users. A new endpoint was implemented that allows users to request its encrypted logs. Based on the stored metadata, the server can filter all logs concerning this particular user. A second endpoint allows logged-in users to fetch logs, which were shared with them by other users. Moreover, the data owner of a log can update a log to share or revoke access. Thus, a third endpoint allows data owners to modify stored logs.

Since all logs are end-to-end encrypted, the *Safekeeper* cannot access the content of the stored data. This implies that it cannot verify if a performed access was permitted by the data owner. However, the endpoints to maintain access policies still exist. They allow data owners to provide detailed policies about who can access what data. A monitor that tracks data usages and creates logs can check if the observed access is allowed by calling those endpoints. Since the *Safekeeper* cannot validate the compliance of these policies, this validation must be outsourced to the monitor.

[4]`https://www.sqlite.org/json1.html`

# 7 Evaluation

This chapter evaluates the protocol proposed in Chapter 5. Besides the verification of the intended functionality in Section 7.1, the security and performance of the protocol are investigated in Sections 7.2 and 7.3. This evaluation verifies that the system requirements given in Chapter 3 can be fulfilled.

## 7.1 Functionality

This section evaluates the functionality of the designed protocol. Specifically, the protocol must ensure that the data owner can always access its logs. The data owner must also be able to share and revoke access to logs. Please see Section 3.2 for details about those functional requirements.

To ensure that data owners can always access their logs, the protocol relies on three mechanisms: First, the monitor must encrypt a new log only for the data owner. Hence, a log is accessible for the data owner upon creation. Second, whenever the data owner shares or revokes access to a log, it must include its own identity in the set of authorized users. Finally, the server must ensure that no other user besides the data owner can modify encrypted logs. This guarantees that the data owner is always able to download and decrypt its logs.

The protocol is based on hybrid encryption. This means that the log is encrypted with a symmetric encryption scheme. The used symmetric key is then encrypted for each intended recipient. Those encrypted keys are attached to the encrypted log. This allows the construction of ciphers that can be decrypted by multiple users. Hence, a data owner can re-encrypt the log for a specific set of users by applying the encryption algorithm. The specified metadata allows the server to filter logs for the requesting user. Moreover, the server can ensure that only the specified data owner can update a log. This effectively enables the functionality to share and revoke access to the log. Sharing a log is realized by adding additional users to the set of authorized users. Revoking access to logs requires omitting a user from this set. Since new key material is generated during encryption, revoked users cannot re-use older decryption keys to access the log.

This analysis shows that the designed protocol fulfills all identified functional requirements. A data owner can always access its logs and it can share and revoke access to them. The protocol was included in the existing toolchain within the scope of this thesis. Section 6.2 investigates the implemented changes. This integration verifies that the protocol has the intended functionality and can be used in practice. Moreover, the implemented libraries are tested. This creates further confidence in their functionality.

## 7.2 Security

This section investigates the security of the designed protocol. Three security requirements were identified during requirements engineering (details in Section 3.3). Each requirement was motivated by a concrete attack scenario. In the following sections, the proposed protocol is confronted with those attacks. They argue why the established security mechanisms resist them.

### 7.2.1 Assumptions

The security of the protocol relies on three fundamental assumptions. If one of those assumptions is violated, the protocol is considered insecure.

First, it is assumed that the protocol has access to a secure PKI. The PKI assigns the identities of users to their public keys via certificates. The user must have exclusive access to his private keys. The public keys are distributed via certificates, which are signed by a trusted certificate authority. If any private key is compromised, the intended E2EE is broken. A private key is assumed to be compromised if any entity besides the owner of the private key has access to it.

Second, it is assumed that the JOSE standard provides secure encryption and signing algorithms. Specifically, `A256GCM` and `ECDH-ES+A256KW` must be secure encryption algorithms and `ES256` must be a secure signing algorithm. This implies that it is computationally infeasible to break those algorithms without knowing the respective keys [28].

Third, the libraries implemented in this thesis rely on language-specific JOSE implementations (details in Section 6.1). It is assumed that those libraries follow the specifications and definitions of the JOSE protocol.

### 7.2.2 Curious server

A curious server is a passive attacker that tries to access the logs within the toolchain. It motivated security requirement S1, which states that only authorized users can access logs. Recall the attacker Eve who is the database admin of the *Safekeeper* server (details in Section 3.3). This section argues why Eve cannot access logs if he was not explicitly authorized.

The protocol enforces that only encrypted data is stored in the database. Both, the monitor creating a log and the data owner sharing a log must encrypt the data before sending it to the server. The server only has access to the JWE token created by the encryption algorithm. This JWE token relies on hybrid encryption: First, a symmetric key $k$ is randomly generated and the data is encrypted via `A256GCM`. This is an authenticated[1] symmetric encryption algorithm based on AES [19]. Secondly, this symmetric key $k$ is encrypted for each receiver using `ECDH-ES+A256KW` [19]. This key-wrapping algorithm establishes an ephemeral key between two users. This ephemeral key is finally used to encrypt the symmetric key $k$. Details about this algorithm can be found in Section 5.4.

To access a decrypted log, the attacker Eve needs to have access to the symmetric key $k$. This requires him to decrypt any of the encrypted keys attached to the JWE token. To

[1] Note that this authentication does not guarantee the authenticity of the sending user. This is because in a symmetric encryption scheme the authenticity is always bound to the knowledge of the symmetric key. Since multiple receivers might have access to the symmetric key, this authentication only ensures that any of those users generated the cipher. To ensure the authenticity of the sending user, the protocol must include a digital signature. [9, p. 315]

succeed, Eve must have access to any of the ephemeral keys used to encrypt $k$. Eve can compute an ephemeral key only if he has access to either the ephemeral private key of the sender or the static private key of the receiver. However, those private keys must be kept secret. If Eve wants to decrypt the log, he must either break the encryption algorithms or access a private key. This, however, contradicts the assumptions. If Eve can break the encryption, the used encryption algorithm is not secure. If Eve can access a private key of a user, the PKI is not secure. This leads to the conclusion that the malicious database admin Eve cannot access decrypted logs. Only if he was explicitly authorized, he can restore the symmetric key $k$, which allows the decryption of a log.

Note that a curious server has access to the metadata of an encrypted log. This metadata consists of the identity of the data owner, which ensures only authorized users can update the stored log, and the identities of the recipients, which ensures that the *Safekeeper* can distribute the logs to the correct users. Although intermediate servers cannot access the log data itself anymore, the metadata yields the information on which users have access to which log. This information could be subject to a metadata analysis attack [5, 6]. If the underlying single-sign-on server supports pseudonyms for users, however, this attack could be made significantly more difficult. Pseudonyms could be assigned to the users participating in the system. Instead of providing a unique identity within the metadata, a random pseudonym of the user could be chosen. Assuming that the intermediate server cannot link pseudonyms back to users, this would make the metadata analysis pointless.

### 7.2.3 Surreptitious forwarding

Surreptitious forwarding refers to an attack where a malicious user re-encrypts and forwards a received log for unintended users [32]. Since such attacks must be detected, the security requirement S2 was introduced. Receivers of encrypted logs must be able to verify that the log was intentionally shared with them. Recall the attacker described in Section 3.3. Alice shares a log with Eve. This implies that Eve has access to the decrypted data. Eve, however, does not stick to the protocol and encrypts the received log for Bob. Bob must be able to detect this fraud because the log he received was not encrypted by the data owner Alice. This section argues why the designed protocol can detect this attack.

The construction introduced to defend against this attack is the *shared log*. It is a JWS token that is signed by the creator of the log. For details see the visualization in Figure 5.2. A *shared log* contains the log as a nested JWS token. Moreover, it contains the identity of the creator and the identities of the intended receivers. Whenever a user successfully decrypts a log, it must verify that the *shared log* is a valid data structure (see Section 5.5 for details). This includes the following three checks. First, the receiver needs to verify if the *shared log* was signed by the claimed creator. Second, the receiver then needs to check if the claimed creator is also the data owner of the log. Third, the receiver must verify if his identity is specified in the list of intended receivers. The following consequence is observed from this validation: The creator specified in the *shared log* must have signed the *shared log*. The creator specified in the *shared log* must be equal to the data owner. Thus, the *shared log* is only valid if the data owner has created the signed log.

Eve has two options to forward an encrypted log to the unintended receiver Bob. First, Eve could manipulate the identity of the data owner. If he manages to change the identity of the data owner to its own identity, he could correctly sign the *shared log*. This modification, however, makes his attack pointless. Eve does not want to introduce faked logs. He wants to forward existing logs to unintended receivers. Moreover, this requires Eve to be a valid monitor because only they are allowed to sign logs. Second, Eve could try to forge the signature of the data owner Alice. This does not require him to modify the log. Rather, Eve keeps the existing log and creates a *shared log* that includes Bob as a valid receiver. He also specifies Alice as the creator of the *shared log*. This approach requires Eve to create a signature in the name of Alice because Bob will check if the claimed creator signed the *shared log*. However, this contradicts the assumptions: To successfully forge a signature, Bob either has access to the private signing key of the data owner or he breaks the signing algorithm. The former breaks the security of the assumed PKI. The latter breaks the security of the chosen signing algorithm.

This analysis shows that there is no way for Eve to forward a received log to unintended recipients if those recipients validate the *shared log* correctly.

### 7.2.4 Malicious data owner

A malicious data owner is a user trying to modify existing logs or create arbitrary new logs. It then tries to share the forged log with other users in the system. Security requirement S3 was introduced to defend against this attack. This section elaborates on how recipients of logs can detect this fraud.

The designed protocol defines a log as a cryptographically signed data structure. Each log must be signed by a valid monitor. The protocol also requires the validation of this signature during decryption. In particular, a receiving user must perform two checks to ensure the validity of the log. First, the monitor specified within the log must have signed the log. This requires the receiver to download the public verification key of the claimed monitor. The receiver can be sure that the log was signed by the private key of the monitor if the verification succeeds. Second, the receiving user must also validate that the claimed monitor is allowed to create logs. Otherwise, arbitrary users could sign logs. This requires the installed PKI to deliver this information, e.g. by encoding it into the certificates of the monitor components.

Consider Eve to be a malicious data owner trying to share a forged log. First of all, Eve creates his malicious log data. He then needs to sign this data using the private signing key of the monitor specified in the log. Otherwise, the recipient would notice that the claimed monitor did not sign the log. Eve has two options. First, he could specify his identity and sign the log with his private signing key. Eve must be a valid monitor because otherwise, the recipient would detect the fraud. Second, Eve could specify an existing monitor in the log. This passes the check if the claimed monitor is allowed to create logs. However, it requires Eve to create a signature in the name of the monitor, which contradicts the assumptions. Either he needs access to the private signing key of the claimed monitor (insecure PKI) or he needs to break the signing algorithm (insecure algorithm). Both options are not feasible under the given assumptions. Thus, a malicious data owner Eve cannot introduce forged logs into the system.

## 7.3 Performance

This section contains performance evaluations of the provided implementations. Section 7.3.1 analyzes the encryption duration of the implemented cryptographic libraries. Section 7.3.2 investigates the overhead of the added encryption layer in the *transparency toolchain*.

### 7.3.1 Encryption

As elaborated in the previous chapters, the designed and implemented protocol is based on hybrid encryption. Hence, logs can be encrypted for multiple recipients. The symmetric key $k$, which encrypts the log, must be encrypted for each receiver. This implies that if a log is shared with $n$ users, the key $k$ must be encrypted $n$ times. Thus, the encryption duration is assumed to increase linearly with the number of recipients. This section aims to provide resilient measurements of those costs.

#### Methodology

The encryption times are measured for each library separately (ts-it-crypto, py-it-crypto, and go-it-crypto). The measurements are implemented as performance tests in each library. All of them follow the same principles. The encryption duration is quantified by measuring the passed milliseconds during encryption. Always the same log data is encrypted. However, the number of recipients changes during different tests. For a fixed number of recipients, a test is repeated 100 times. The average of those runs is said to be the encryption duration for the number of recipients in the library under test.

The performance tests were executed on a machine running *macOS Monterey* with an *Apple M1* ARM64 CPU and 16 GB RAM. The Go library was compiled with Go 1.18.3[2]. The Typescript package was compiled into Javascript using TSC 4.8.3[3]. It was then executed by the Node.js version 18.12.1[4], which is built upon the V8 engine 10.2.154.15[5]. The Python tests were run under Python 3.9.6[6] using the CPython[7] interpreter.

#### Results

The results of all three libraries are illustrated in Figure 7.1. It shows the encryption duration in milliseconds for a different number of recipients. In general, one can observe that the Go implementation is the fastest library. This meets the expectations because

[2] `https://go.dev/doc/devel/release`
[3] `https://www.typescriptlang.org/docs/handbook/compiler-options.html`
[4] `https://nodejs.org/ca/blog/release/v18.12.1/`
[5] `https://v8.dev/`
[6] `https://python.org/downloads/release/python-396/`
[7] `https://github.com/python/cpython`

Go is a compiled language while Python and Typescript are interpreted. Note that Typescript is compiled into Javascript. The resulting Javascript code, however, is also interpreted by the V8 engine included in Node.js. The performance difference between the Python and Typescript implementations is remarkable. Both rely on the native OpenSSL module installed on the operating system. Hence all complex cryptographic tasks are computed by pre-compiled libraries. This improves the performance of both libraries. The observed speed difference can be explained best when considering the performance differences between the Javascript engine V8 and the CPython interpreter: The experiments performed by [56] indicate that execution times under V8 are on average almost four times faster than under CPython.

For all packages, one can observe a linear growth of encryption times when adding additional recipients. This meets the expectations because all libraries rely on hybrid encryption. When adding a new user, the encrypting party must additionally encrypt the secret key for this user. The encryption for a single receiver in Go takes $0.12ms$. The encryption for two receivers takes $0.17ms$. The additional receiver increases the costs by about $0.05ms$ because both runs encrypted the same log. The same analysis can be done for the Python and Typescript libraries. While an additional recipient in Python increases the encryption costs by approximately $0.50ms$, the Typescript implementation requires about $0.30ms$ for an additional user.

From the perspective of a user, a system reacts instantaneously if a computation takes less than $100ms$ [57]. In those cases no visual feedback is necessary. Experiments with the packages show that this threshold is reached in Typescript if encrypting the log for about 330 users. The Python implementation reaches the boundary if data is encrypted for 170 users. In Go, the encryption for 1500 recipients takes on average slightly more than $100ms$. This shows that a data owner can share a log with up to 170 users without noticing a computation delay during encryption.

### 7.3.2 Toolchain

The *transparency toolchain* was adopted to support encrypted logs. Once a user logs in to the *Display* component, all encrypted logs of this particular user are fetched from the *Safekeeper*, decrypted, and visualized. Compared to the legacy solution, this implementation introduces an additional encryption layer. It is assumed that this degrades the performance of fetching and visualizing logs because all received logs must be decrypted in the front-end. This section provides measurements of 1) the legacy toolchain that does not rely on encryption and 2) the updated toolchain that handles encrypted logs. Those measurements illustrate the performance differences of the toolchain when encrypted logs are fetched from the server and processed in the front-end.

#### Methodology

The measurements were conducted within the *Display* and the *Safekeeper* component. They reflect the passed time from starting an API call to fetch logs until those logs are ready for visualization. This time can be divided into three sections: Database queries,

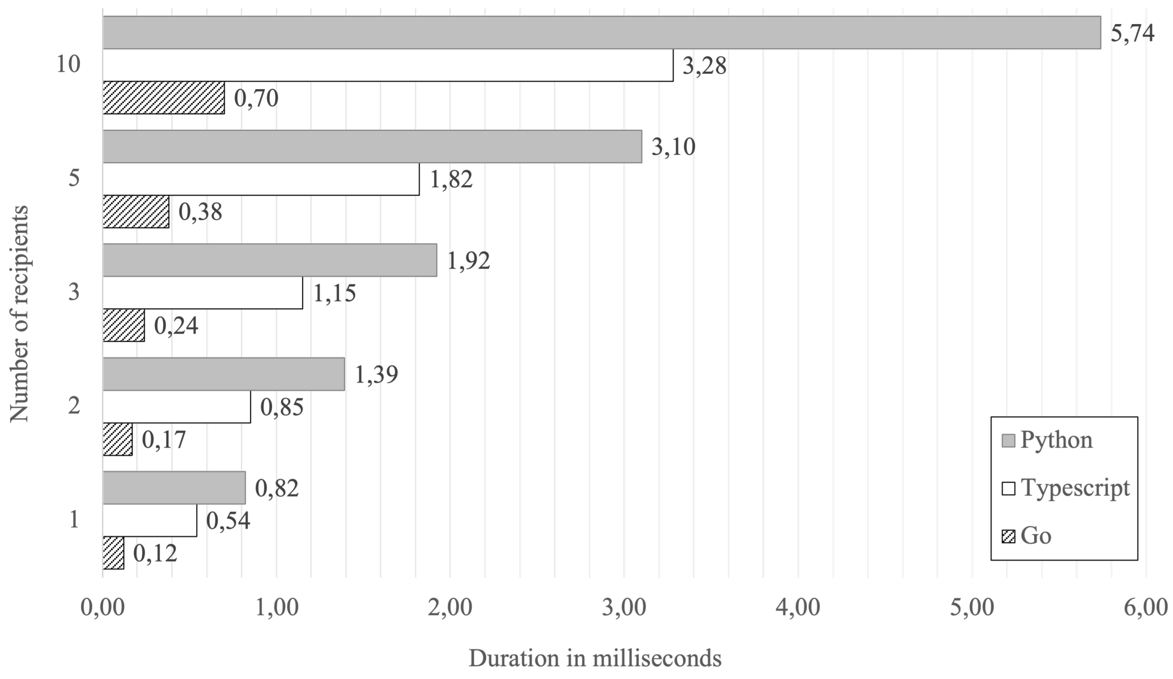


**Figure 7.1:** *This figure visualizes the encryption duration of the different libraries in milliseconds. The same log data was encrypted for a different number of recipients.*

server computation time, and client computation time. First, the database queries reflect the time the server needs to extract the logs and other information from the database. Second, the server computation time is the time the server needs to resolve the API call besides the time for querying the database. For example, the legacy toolchain computes some metadata data over the returned logs and includes this data in the response. Third, the client computation time is the time needed by the client to process the fetched data until it is available for visualization. For example, the overview data must be computed by the front-end in the context of encrypted logs. This is because this metadata can only be calculated once the logs are decrypted. Moreover, the client computation time includes all tasks that are performed once the API call has finished. The time for decrypting logs is also part of the client computation time.

The database queries are measured directly within the *Safekeeper* component. The *Display* component measures the time needed for the entire API call. To compute the server computation time, the database queries are subtracted from the duration of the entire API call. The client computation time is measured directly within the *Display* component.

To provide resilient measurements, the database always contains 1000 logs for the logged-in user. The number of fetched and visualized logs changes during different tests. The tests run for 1, 10, and 100 logs. For a fixed number of logs, a test is repeated 100 times. The presented numbers are the averages over all runs. Each test is run against the legacy toolchain and the updated toolchain. The tests were executed within the development environment, which was started using `docker-compose`. They were executed on a machine running *macOS Monterey* with an *Apple M1* ARM64 CPU and 16 GB RAM.

**Results**

Figure 7.2 shows the results of the legacy toolchain. Overall, the toolchain requires on average 142*ms* to fetch a single log, 148*ms* to fetch ten logs, and 243*ms* to fetch one hundred logs. The server computation time increases when multiple logs are fetched. This is expected because the server computation time includes the transmission delay over the network. Furthermore, one can observe that the client computation time is very fast. The main reason for this lies in the fact that the server performs different pre-computation steps needed to visualize the logs in the front-end. Hence, the front-end can directly visualize the logs without further computation. For example, the server fetches metadata from the database to provide the number of logs that were created within a specific tool. The database queries account for a big portion of the time needed to fetch logs from the server. There are two reasons for this. First, the pre-computed data relies on multiple database queries. Second, the realized database schema requires a database join to resolve the identity of a user to a given access log. Thus, the measured database query times are higher in the legacy toolchain than in the updated toolchain. The time needed to query data from the database increases slightly when more logs are loaded from the server. This is expected since the database must resolve more data if more logs are requested.

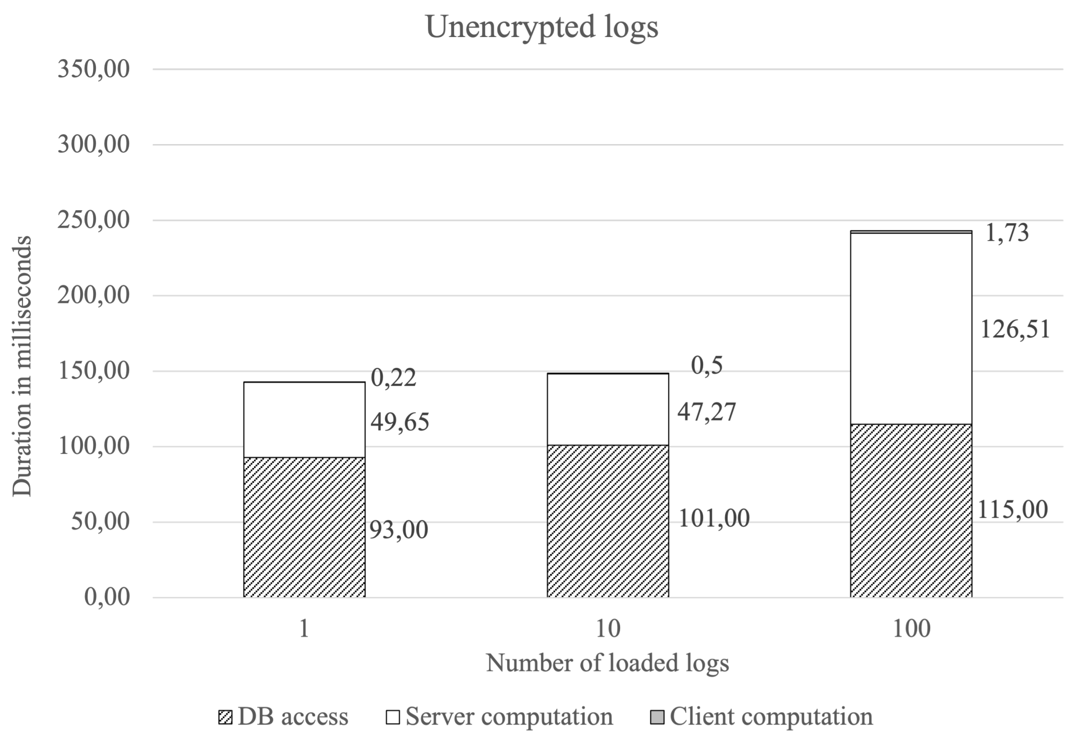


**Figure 7.2:** *This figure visualizes the measured times to download and process the logs in the previous toolchain, which maintains unencrypted logs.*

Figure 7.3 shows the results of the updated toolchain including the encryption layer. Overall, the updated toolchain requires on average 77*ms* to fetch a single log, 106*ms* to fetch ten logs, and 298*ms* to fetch one hundred logs. Those results are not expected because for a single log and ten logs the encrypted toolchain is faster than the legacy toolchain. The reason for this lies in the pre-computation performed by the legacy toolchain, which depends on multiple database queries. Those pre-computation queries require more time than the decryption of the logs in the updated toolchain. Moreover, the updated toolchain employs a simplified database design consisting of a single table that stores the encrypted logs. As described in Section 6.2.3, this table contains two JSON fields. They could be further improved to support efficient indexed lookups [55]. Querying the logs from this table is more efficient than querying logs in the old database scheme because only a single query without database joins is executed. Thus, the aggregated duration of all database queries is smaller in the updated toolchain compared to the legacy version. Figure 7.3 illustrates that server computation times increase if multiple logs are fetched. Again, this is expected because it contains the transmission delay of the data sent over the network. The more data is transferred, the longer the transmission delay. Compared to the legacy toolchain, the server computation time has increased. The size of an encrypted log is about 1600 bytes. The size of an unencrypted log is 700 bytes on average. The updated toolchain must transfer more data than the legacy version because encrypted logs introduce a memory overhead. Hence, the legacy toolchain requires less server computation time. Finally, the client computation time is higher in the updated toolchain. If the client fetches 100 logs, their processing takes about 114*ms*. Compared to the very short processing times in the legacy version, this is a significant increase. There are two reasons for this. First, the received logs must be decrypted. Second, the decrypted logs are analyzed to provide metadata for the visualization in the front-end. In the legacy version, this was computed by the server. Since the log data must be decrypted to compute this metadata, however, this computation was shifted to the client. Hence, the decryption of logs and their processing is considered to be the bottleneck of the updated toolchain. If 100 logs are fetched from the server, the updated toolchain is 55*ms* slower than the legacy toolchain. This is a relative difference of 20%.

This analysis delivers two major findings. First, the updated toolchain enables faster database queries due to the simplified database scheme. This implies that the legacy version is even slower if a small number of logs are requested. Second, the handling of encrypted logs introduces a significant overhead on the client side if many logs are fetched. This bottleneck should be avoided by not fetching more than 100 logs at once. Note that the front-end makes use of pagination. This analysis shows that the page size should not exceed 100 logs. If only 10 logs are loaded per page, the updated toolchain is even faster than the legacy version.

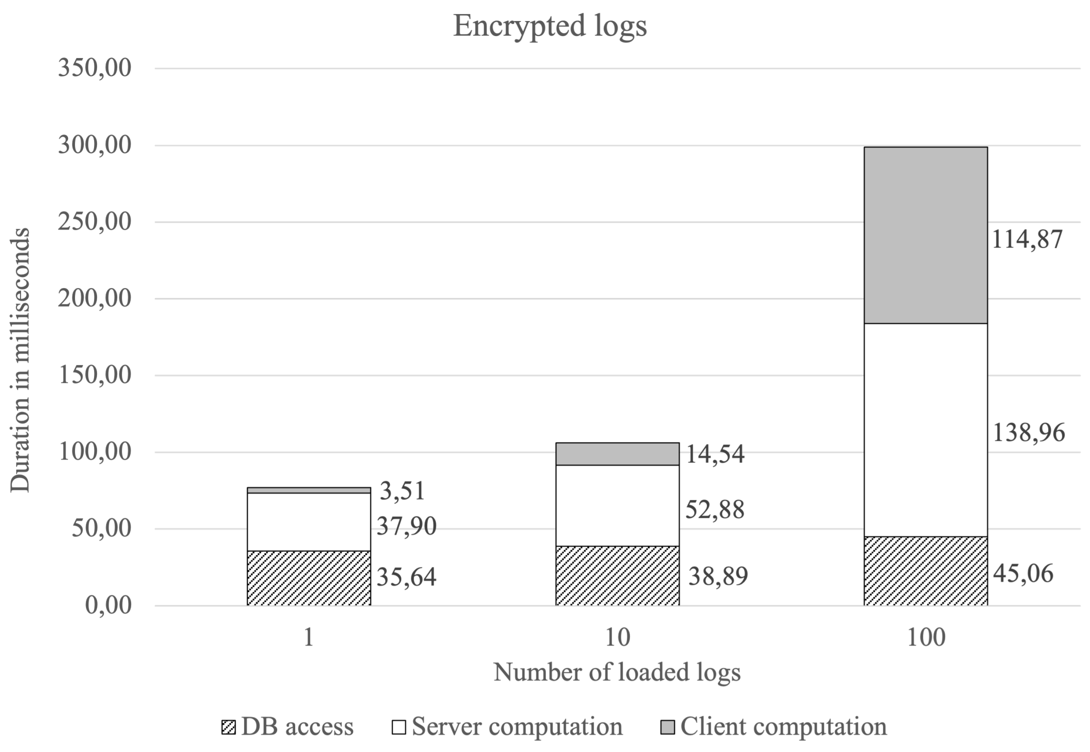


**Figure 7.3:** *This figure visualizes the measured times to download and process the logs in the updated toolchain, which maintains encrypted logs.*

# 8 Conclusion and Discussion

This final chapter concludes the thesis. It summarizes the major results of the thesis in Section 8.1. It also discusses the limitations and potential future work in Section 8.2.

## 8.1 Results

This thesis designs, implements, and evaluates an end-to-end encrypted protocol that allows to share logs in the *transparency toolchain* [3]. It identifies the requirements that must be fulfilled by such a protocol. The survey of encryption technologies shows that hybrid encryption meets those requirements best. Therefore, the designed protocol relies on hybrid encryption. The protocol was created to resist three types of attackers: A curious server cannot access the logs because all logs are encrypted. Surreptitious forwarding attacks are detected since the user sharing a log cryptographically signs the log along with the indented receivers. A malicious data owner cannot forge logs because each log must be cryptographically signed by a trusted monitor component.

The protocol was implemented in the programming languages Go, Python and Typescript. For each of those, a library was developed, which can be used to sign, encrypt and decrypt logs. These libraries enable the intended functionality of a multi-party E2EE protocol. Moreover, each library is tested, documented, and published to the respective package indexes.

The existing toolchain was adapted to enable the functionality of sharing encrypted logs. This is realized by including the implemented library in the toolchain. A proof-of-concept environment demonstrates how an established PKI may be used to instantiate the protocol. This verifies that the protocol can be used in practice to share encrypted logs among users. From the perspective of a user, the encrypted logs can be visualized as before. Access to those logs can additionally be shared and revoked. The implemented UI fully abstracts the underlying cryptographic details such as digital signatures, encryption, and decryption algorithms.

The integration of the designed protocol into the toolchain verifies that the functional requirements can be fulfilled. The security evaluation shows that the protocol can be considered secure under two assumptions. First, a secure PKI must be available. Second, the JOSE standard must describe secure algorithms. The performance evaluation shows that the protocol introduces overhead due to the added encryption layer. The encryption algorithm does not introduce a human-noticeable delay if the set of receivers does not contain more than 170 users. The bottleneck of the encryption layer occurs when multiple logs are downloaded from the server because this requires the decryption of all loaded logs. If the front-end restricts the page size of the fetched logs to 100, the processing time in the updated toolchain has increased by $55ms$ to $298ms$. This indicates that the introduced overhead is kept within reasonable limits and can be handled in practice.

In conclusion, the designed and implemented protocol maintains end-to-end encrypted logs, which can be shared among users. This protects the confidentiality of the stored logging data. It also supports data owners that aim to prosecute illegal data accesses.

## 8.2 Limitations and future work

This thesis is designed to defend against three types of attacks. They were derived from potentially malicious users participating in the sharing process of encrypted logs. This, however, is not an exhaustive list. The performed security analysis does not guarantee the general security of the protocol. In particular, a third-party analysis of the applied techniques could further verify their security. This could also include sophisticated mathematical proofs of the protocol as introduced by [28]. The security evaluation applied in this thesis at least indicates that the protocol can resist the assumed attackers if the respective assumptions are true.

The protocol tries to encrypt the logs because they might contain sensitive information. However, the encrypted logs also reveal metadata to ensure that the encrypted data can be associated with users by intermediate servers. This metadata could be subject to metadata analysis attacks, which might yield interesting information for an attacker. Future work could investigate if this is a real threat. The approach of encrypting logs still improves the current situation because it keeps the content of logs confidential.

This thesis assumes a secure PKI meaning that nobody knows the private keys of the users. This is a strong assumption. If the PKI is established by a trusted entity knowing the private keys of the users, the strict interpretation of E2EE is broken. This is because in such a case not only the authorized users can decrypt the logs. The knowledge of the private keys allows the trusted entity to decrypt and sign arbitrary data. The protocol might be improved in the future for scenarios where the assumption of a secure PKI does not hold.

# List of Figures

# List of Listings

# List of Tables

# Bibliography

[1] A. Boes, T. Hess, A. Pretschner, T. Kämpf, and E. Vogl. *Daten–Innovation–Privatheit–Mit Inverser Transparenz das Gestaltungsdilemma der digitalen Arbeitswelt lösen*. 2022. eprint: `https://doi.org/10.36194/INVERSETRANSPARENZ_Mai_2022`.

[2] D. R. Schallmo. "Digitale Transformation von Geschäftsmodellen". In: *Jetzt digital transformieren*. Springer, 2016. eprint: `https://doi.org/10.1007/978-3-658-14569-9_2`.

[3] V. Zieglmeier and A. Pretschner. "Trustworthy transparency by design". In: *arXiv:2103.10769* (2021). eprint: `https://doi.org/10.48550/arXiv.2103.10769`.

[4] E. Rescorla. *HTTP over TLS*. Tech. rep. 2000. eprint: `https://www.rfc-editor.org/rfc/rfc2818`.

[5] B. Greschbach, G. Kreitz, and S. Buchegger. "The devil is in the metadata — New privacy challenges in Decentralised Online Social Networks". In: *IEEE International Conference on Pervasive Computing and Communications Workshops*. 2012. eprint: `https://doi.org/10.1109/PerComW.2012.6197506`.

[6] J. Mayer, P. Mutchler, and J. C. Mitchell. "Evaluating the privacy properties of telephone metadata". In: *Proceedings of the National Academy of Sciences* 20 (2016). eprint: `https://doi.org/10.1073/pnas.1508081113`.

[7] A. Weber. *Nutzung von Cloud-Computing steigt im Corona-Jahr*. June 2021. eprint: `https://www.bitkom-research.de/de/pressemitteilung/nutzung-von-cloud-computing-steigt-im-corona-jahr`.

[8] K. Mallory, F. Baker, K. Olaf, C. Sofia, and G. Gurshabad. *Definition of End-to-end Encryption*. July 2022. eprint: `https://datatracker.ietf.org/doc/draft-knodel-e2ee-definition/`.

[9] C. Eckert. *IT-Sicherheit*. De Gruyter Oldenbourg, 2018. ISBN: 9783110563900. eprint: `https://doi.org/10.1515/9783110563900`.

[10] M. Wolf, A. Weimerskirch, and T. Wollinger. "State of the art: Embedding security in vehicles". In: *EURASIP Journal on Embedded Systems* (2007). eprint: `https://doi.org/10.1155/2007/74706`.

[11] J. H. An, Y. Dodis, and T. Rabin. "On the security of joint signature and encryption". In: *International conference on the theory and applications of cryptographic techniques*. Springer. 2002. eprint: `https://doi.org/10.1007/3-540-46035-7_6`.

[12] R. Barnes. *Use Cases and Requirements for JSON Object Signing and Encryption (JOSE)*. Tech. rep. 2014. eprint: `https://www.rfc-editor.org/info/rfc7165`.

[13] S. Frankel and S. Krishnan. *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. Tech. rep. 2011. eprint: `https://www.rfc-editor.org/rfc/rfc6071`.

[14] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Tech. rep. 2018. eprint: `https://www.rfc-editor.org/rfc/rfc8446`.

[15] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Signature (JWS)*. Tech. rep. 2015. eprint: `https://www.rfc-editor.org/rfc/rfc7515`.

[16] M. Jones and J. Hildebrand. *JSON Web Encryption (JWE)*. Tech. rep. 2015. eprint: `https://www.rfc-editor.org/rfc/rfc7516`.

[17] M. Jones. *JSON Web Key (JWK)*. Tech. rep. 2015. eprint: `https://www.rfc-editor.org/rfc/rfc7517`.

[18] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. Tech. rep. 2015. eprint: `https://www.rfc-editor.org/rfc/rfc7519`.

[19] M. Jones. *JSON Web Algorithms (JWA)*. Tech. rep. 2015. eprint: `https://www.rfc-editor.org/rfc/rfc7518`.

[20] C. Gentry. "Certificate-Based Encryption and the Certificate Revocation Problem". In: *Advances in Cryptology — EUROCRYPT 2003*. Springer Berlin Heidelberg, 2003. eprint: `https://doi.org/10.1007/3-540-39200-9_17`.

[21] R. Sakai and J. Furukawa. "Identity- Based Broadcast Encryption." In: *Cryptology ePrint Archive* (2007). eprint: `https://ia.cr/2007/217`.

[22] J. Bethencourt, A. Sahai, and B. Waters. "Ciphertext-Policy Attribute-Based Encryption". In: *IEEE Symposium on Security and Privacy*. 2007. eprint: `https://doi.org/10.1109/SP.2007.11`.

[23] J. Hagg. "Literature research: A cryptographic protocol to maintain confidentiality when storing multi-owner data". Studienarbeit. Technical University of Munich, 2022. eprint: `https://mediatum.ub.tum.de/1692382`.

[24] M. Nieles, K. Dempsey, V. Y. Pillitteri, et al. "An introduction to information security". In: *NIST special publication* 12 (2017). eprint: `https://doi.org/10.6028/NIST.SP.800-12r1`.

[25] K. Ermoshina, F. Musiani, and H. Halpin. "End-to-end encrypted messaging protocols: An overview". In: *International Conference on Internet Science*. Springer. 2016. eprint: `https://doi.org/10.1007/978-3-319-45982-0_22`.

[26] B. Hale and C. Komlo. *On End-to-End Encryption*. Cryptology ePrint Archive. 2022. eprint: `https://eprint.iacr.org/2022/449`.

[27] M. Nabeel. "The Many Faces of End-to-End Encryption and Their Security Analysis". In: *2017 IEEE International Conference on Edge Computing (EDGE)*. June 2017. eprint: `https://doi.org/10.1109/IEEE.EDGE.2017.47`.

[28] J. Katz and Y. Lindell. *Introduction to modern cryptography*. CRC press, 2020. eprint: `https://doi.org/10.1201/b17668`.

[29] N. R. Tousif ur Rehman Muhammad Naeem Ahmed Khan. "Analysis of requirement engineering processes, tools/techniques and methodologies". In: *International Journal of Information Technology and Computer Science (IJITCS)* 3 (2013). eprint: `https://doi.org/10.5815/ijitcs.2013.03.05`.

[30] B. Fabian, S. Gürses, M. Heisel, T. Santen, and H. Schmidt. "A comparison of security requirements engineering methods". In: *Requirements engineering* 1 (2010). eprint: `https://doi.org/10.1007/s00766-009-0092-x`.

[31] J. Mylopoulos, L. Chung, and B. Nixon. "Representing and using nonfunctional requirements: a process-oriented approach". In: *IEEE Transactions on Software Engineering* 6 (1992). eprint: `https://doi.org/10.1109/32.142871`.

[32] D. Davis. "Defective Sign & Encrypt in S/MIME, PKCS# 7, MOSS, PEM, PGP, and XML." In: *USENIX Annual Technical Conference, General Track*. 2001. eprint: `https://theworld.com/~dtd/sign_encrypt/sign_encrypt7.html`.

[33] M. Watson. *Web Cryptography API*. 2017. eprint: `https://www.w3.org/TR/WebCryptoAPI/`.

[34] H. Halpin. "The W3C Web Cryptography API: Motivation and Overview". In: *Proceedings of the 23rd International Conference on World Wide Web*. WWW '14 Companion. Association for Computing Machinery, 2014. eprint: `https://doi.org/10.1145/2567948.2579224`.

[35] C. Braz, A. Seffah, and D. M'Raihi. "Designing a Trade-Off Between Usability and Security: A Metrics Based-Model". In: *Human-Computer Interaction – INTERACT*. 2007. eprint: `https://doi.org/10.1007/978-3-540-74800-7_9`.

[36] H. Levenson. *The Principle of Least Effort: An Integral Part of UX*. 2018. eprint: `https://dzone.com/articles/the-principle-of-least-effort-an-integral-part-of`.

[37] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. *OpenPGP message format*. Tech. rep. 2007. eprint: `https://www.rfc-editor.org/rfc/rfc4880`.

[38] J. Schaad, B. Ramsdell, and S. Turner. *Secure/multipurpose internet mail extensions (S/MIME) version 4.0 message specification*. Tech. rep. 2019. eprint: `https://www.rfc-editor.org/rfc/rfc8551`.

[39] L. Seitz, J.-M. Pierson, and L. Brunie. "Key Management for Encrypted Data Storage in Distributed Systems". In: *Second IEEE International Security in Storage Workshop*. 2003. eprint: `https://doi.org/10.1109/SISW.2003.10001`.

[40] A. Sahai, B. Waters, and S. Lu. "Attribute-Based Encryption". In: *IDENTITY-BASED CRYPTOGRAPHY*. Cryptology and Information Security Series. IOS PRESS, 2009. eprint: `https://doi.org/10.3233/978-1-58603-947-9-156`.

[41] N. Vaanchig, H. Xiong, W. Chen, and Z. Qin. "Achieving Collaborative Cloud Data Storage by Key-Escrow-Free Multi-Authority CP-ABE Scheme with Dual-Revocation". In: *International Journal of Network Security* (2018). eprint: `https://doi.org/10.6633/IJNS.201801.20(1).11`.

[42] A. Fiat and M. Naor. "Broadcast Encryption". In: *Annual International Cryptology Conference*. Springer. 1993. eprint: `https://doi.org/10.1007/3-540-48329-2_40`.

[43] J. Li, L. Chen, Y. Lu, and Y. Zhang. "Anonymous certificate-based broadcast encryption with constant decryption cost". In: *Information Sciences* (2018). eprint: `https://doi.org/10.1016/j.ins.2018.04.067`.

[44] C.-I. Fan, P.-J. Tsai, J.-J. Huang, and W.-T. Chen. "Anonymous Multi-receiver Certificate-Based Encryption". In: *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*. 2013. eprint: `https://doi.org/10.1109/CyberC.2013.13`.

[45] A. Shamir. "Identity-Based Cryptosystems and Signature Schemes". In: *Advances in Cryptology*. Springer Berlin Heidelberg, 1985. eprint: `https://doi.org/10.1007/3-540-39568-7_5`.

[46] W.-H. Chen, C.-I. Fan, and Y.-F. Tseng. "Efficient Key-Aggregate Proxy Re-Encryption for Secure Data Sharing in Clouds". In: *IEEE Conference on Dependable and Secure Computing (DSC)*. 2018. eprint: `https://doi.org/10.1109/DESEC.2018.8625149`.

[47] J. Blum, S. Booth, O. Gal, M. Krohn, J. Len, et al. *Zoom Cryptography Whitepaper*. Tech. rep. 2020. eprint: `https://github.com/zoom/zoom-e2e-whitepaper`.

[48] T. Isobe and R. Ito. "Security Analysis of End-to-End Encryption for Zoom Meetings". In: *IEEE Access* (2021). eprint: `https://doi.org/10.1109/ACCESS.2021.3091722`.

[49] H. Xiong, X. Zhang, D. Yao, X. Wu, and Y. Wen. "Towards End-to-End Secure Content Storage and Delivery with Public Cloud". In: *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*. Association for Computing Machinery, 2012. eprint: `https://doi.org/10.1145/2133601.2133633`.

[50] M. Marlinspike and T. Perrin. "The X3DH key agreement protocol". In: *Open Whisper Systems* (2016). eprint: `https://signal.org/docs/specifications/x3dh/x3dh.pdf`.

[51] T. Perrin and M. Marlinspike. "The Double Ratchet Algorithm". In: (2016). eprint: `https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf`.

[52] M. Marlinspike. *Private Group Messaging*. eprint: `https://signal.org/blog/private-groups/`.

[53] E. Barker, E. Barker, W. Burr, W. Polk, M. Smid, et al. *Recommendation for key management*. Tech. rep. National Institute of Standards and Technology, 2006. eprint: `https://doi.org/10.6028/NIST.SP.800-57pt1r5`.

[54] E. Barker, L. Chen, S. Keller, A. Roginsky, A. Vassilev, et al. *Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography*. Tech. rep. National Institute of Standards and Technology, 2017. eprint: `https://doi.org/10.6028/NIST.SP.800-56Ar3`.

[55] S. Shang, Q. Wu, T. Wang, and Z. Shao. “LiteIndex: Memory-Efficient Schema-Agnostic Indexing for JSON Documents in SQLite”. In: *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. ASPDAC ’21. 2021. eprint: `https://doi.org/10.1145/3394885.3431518`.

[56] D. Lion, A. Chiu, M. Stumm, and D. Yuan. “Investigating Managed Language Runtime Performance”. In: *USENIX Annual Technical Conference (USENIX ATC 22)*. 2022. eprint: `https://www.usenix.org/system/files/atc22-lion.pdf`.

[57] J. Nielsen. *Usability engineering*. Morgan Kaufmann, 1994. eprint: `https://www.nngroup.com/articles/response-times-3-important-limits/`.

# A Illustration of algorithms

This appendix illustrates the applied encryption and decryption algorithms. For details about their functionality have a look to Chapter 5.

## A.1 Encryption algorithm

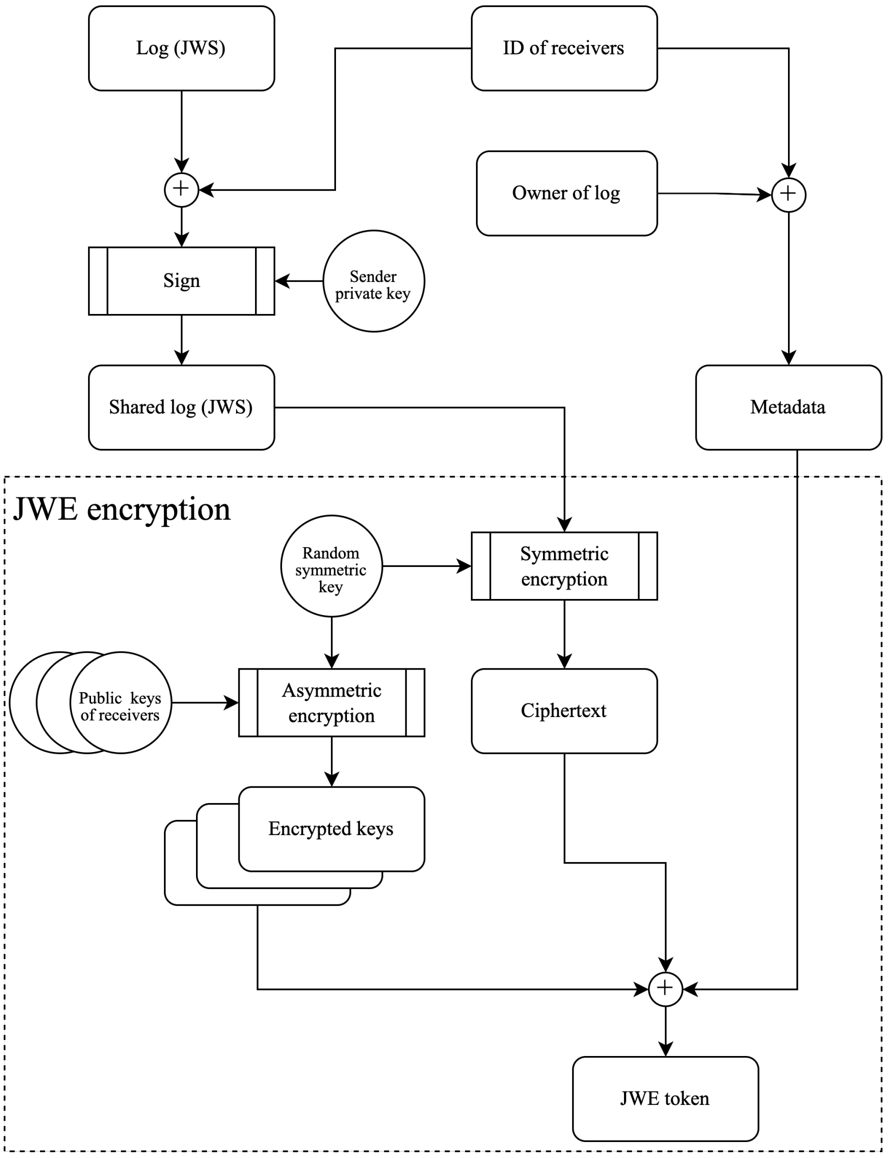


**Figure A.1:** *The encryption algorithm takes a signed log and a set of receivers as input. It returns a JWE token which can only be decrypted by the specified set of users.*

## A.2 Decryption algorithm

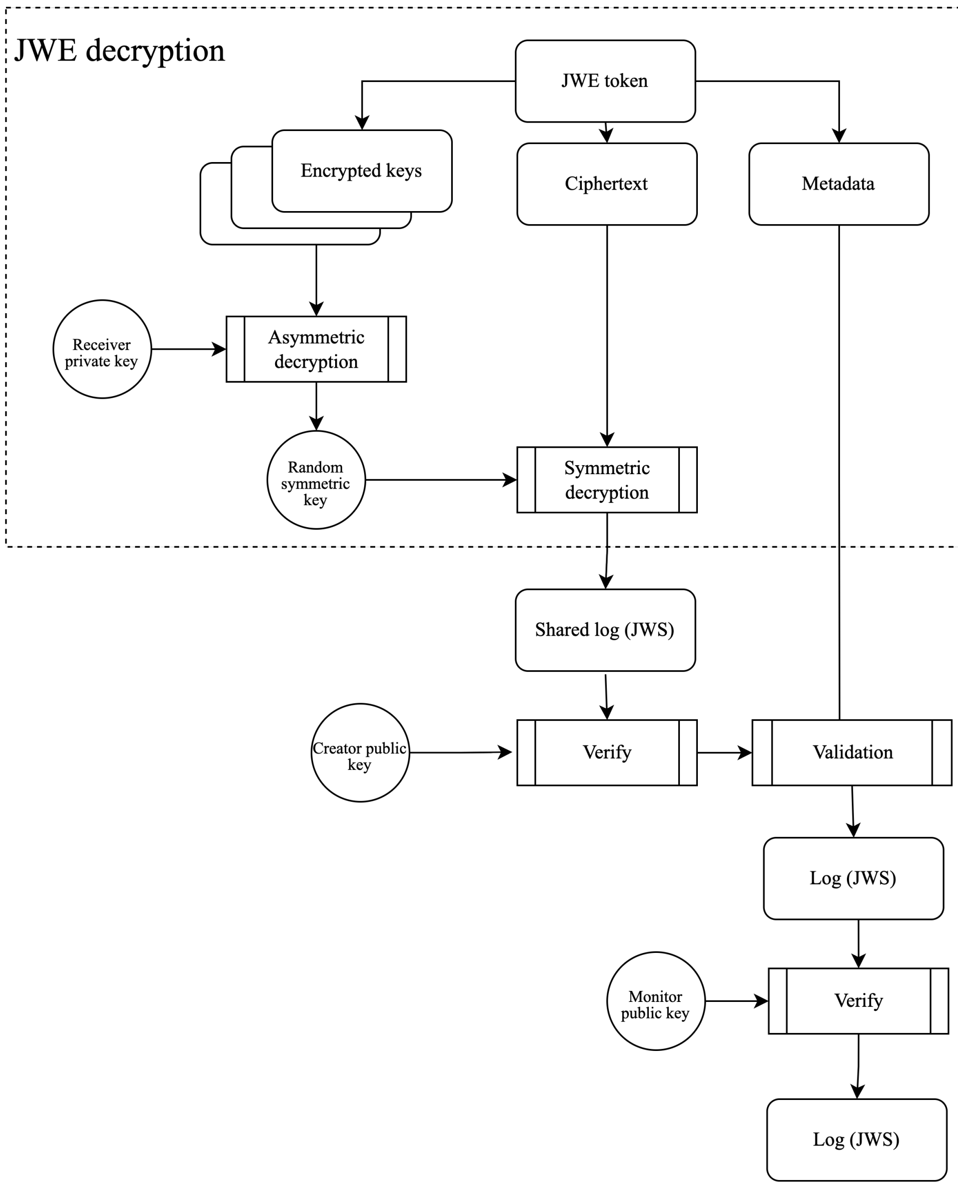
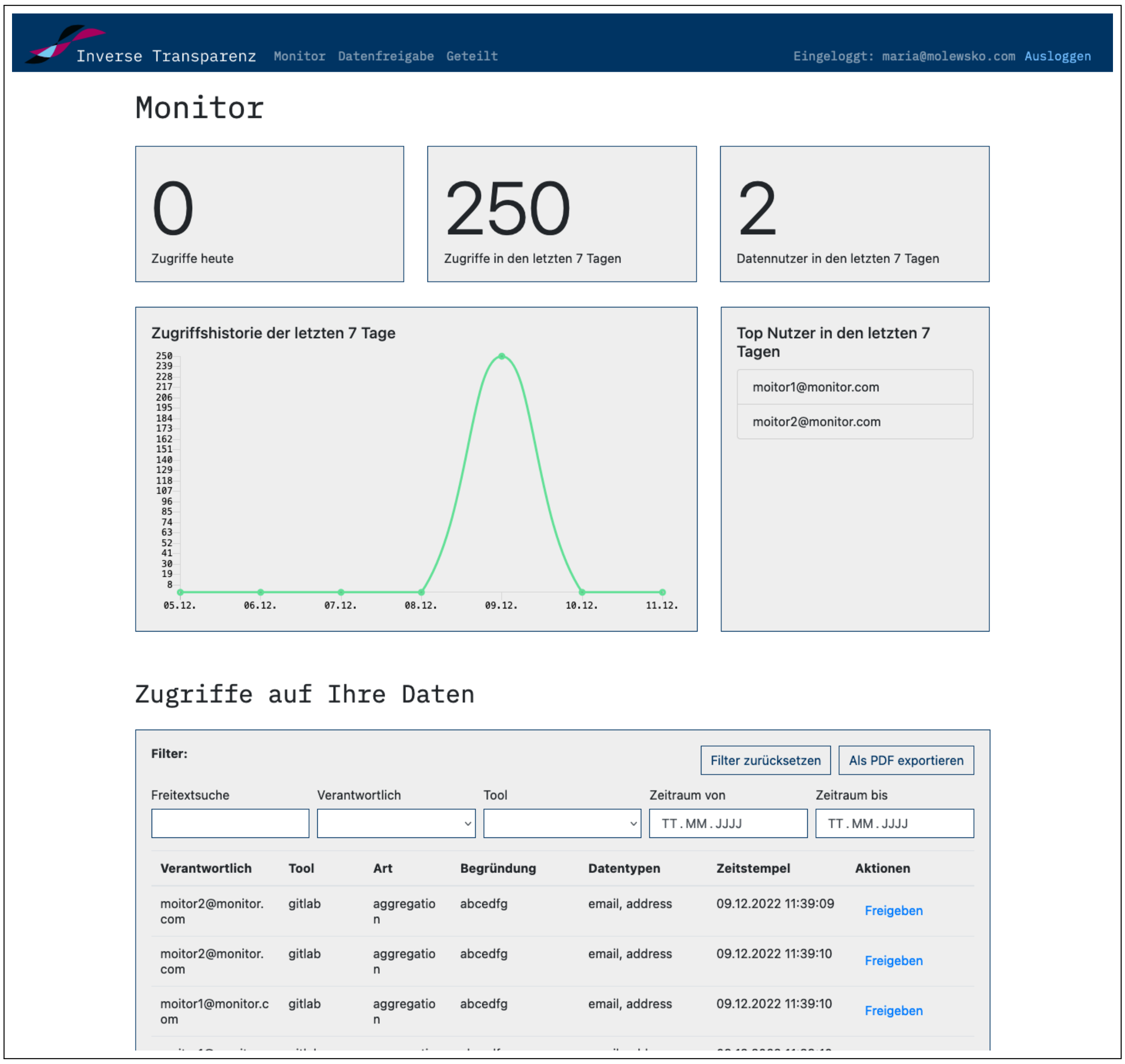


**Figure A.2:** *The decryption algorithm takes a JWE token and a private decryption key as input. It returns the decrypted log if the user is allowed to decrypt.*

# B Screenshots Clotilde UI

This appendix contains screenshots of the implemented features in the *Clotilde UI*.



Monitor

0 Zugriffe heute

250 Zugriffe in den letzten 7 Tagen

2 Datennutzer in den letzten 7 Tagen

Zugriffshistorie der letzten 7 Tage

Top Nutzer in den letzten 7 Tagen

- moitor1@monitor.com
- moitor2@monitor.com

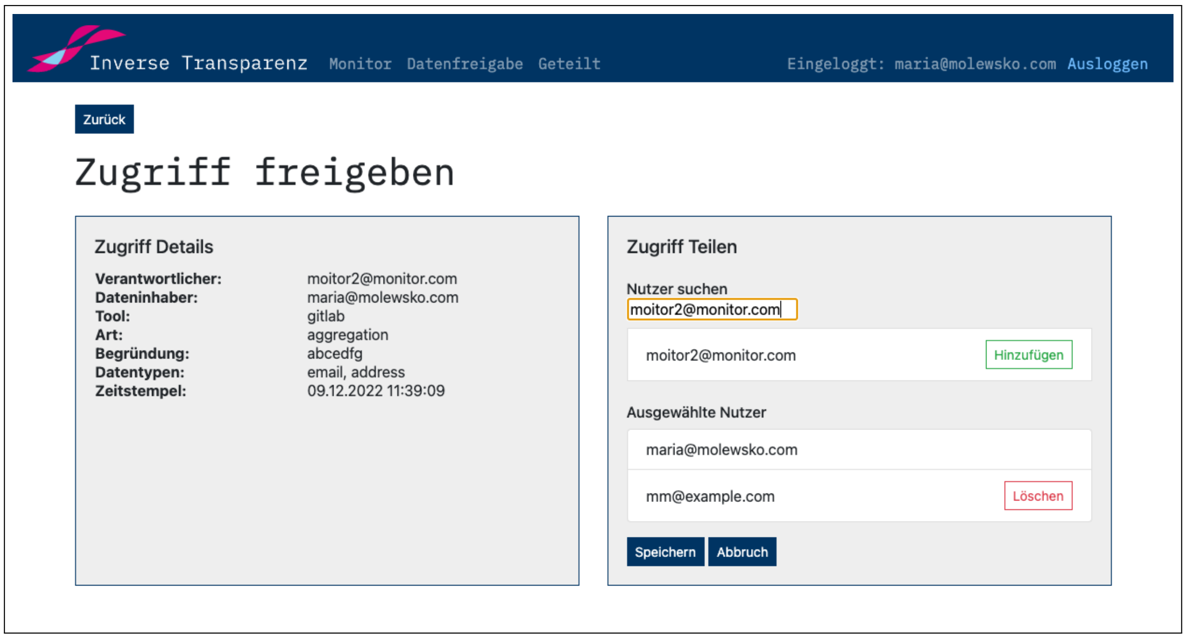
Zugriffe auf Ihre Daten

**Filter:**

| Freitextsuche | Verantwortlich | Tool | Zeitraum von | Zeitraum bis |
|---|---|---|---|---|
| | | | TT . MM . JJJJ | TT . MM . JJJJ |

| Verantwortlich | Tool | Art | Begründung | Datentypen | Zeitstempel | Aktionen |
|---|---|---|---|---|---|---|
| moitor2@monitor.com | gitlab | aggregation | abcedfg | email, address | 09.12.2022 11:39:09 | Freigeben |
| moitor2@monitor.com | gitlab | aggregation | abcedfg | email, address | 09.12.2022 11:39:10 | Freigeben |
| moitor1@monitor.com | gitlab | aggregation | abcedfg | email, address | 09.12.2022 11:39:10 | Freigeben |

**Figure B.1:** *The updated overview page of the Clotilde UI. All logs are visualized as before. The user can share or revoke access to a log by clicking on the corresponding button in the table.*



Zugriff freigeben

Zugriff Details

| | |
|---|---|
| **Verantwortlicher:** | moitor2@monitor.com |
| **Dateninhaber:** | maria@molewsko.com |
| **Tool:** | gitlab |
| **Art:** | aggregation |
| **Begründung:** | abcedfg |
| **Datentypen:** | email, address |
| **Zeitstempel:** | 09.12.2022 11:39:09 |

Zugriff Teilen

Nutzer suchen

moitor2@monitor.com

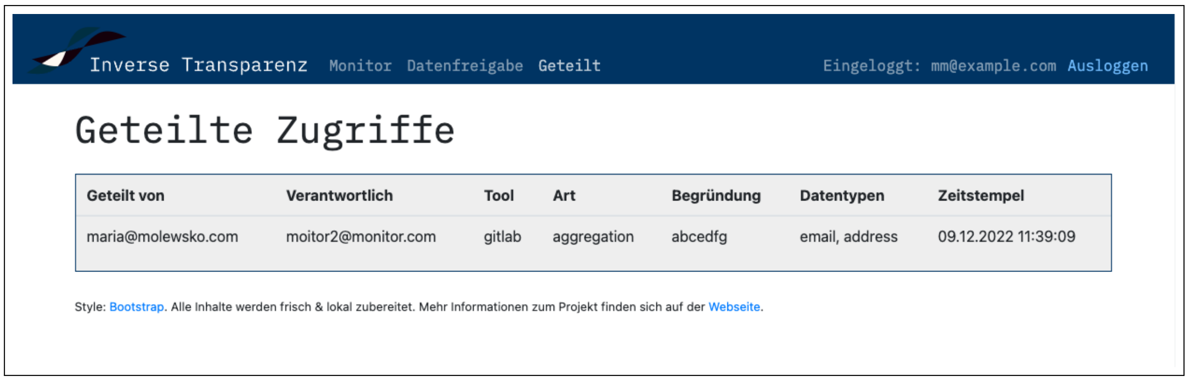
moitor2@monitor.com

Ausgewählte Nutzer

maria@molewsko.com

mm@example.com

**Figure B.2:** *This view allows the user to share or revoke access to a log. On the left-hand side, the details of the log are displayed. On the right-hand side, the user can search for other users in the system. If a user is found, it can be added to the set of recipients. The access to certain users can be revoked by removing them from the list of recipients.*



Geteilte Zugriffe

| Geteilt von | Verantwortlich | Tool | Art | Begründung | Datentypen | Zeitstempel |
|---|---|---|---|---|---|---|
| maria@molewsko.com | moitor2@monitor.com | gitlab | aggregation | abcedfg | email, address | 09.12.2022 11:39:09 |

Style: Bootstrap. Alle Inhalte werden frisch & lokal zubereitet. Mehr Informationen zum Projekt finden sich auf der Webseite.

**Figure B.3:** *This view lists all logs that are shared with the logged-in users. It is realized as additional section within the menu.*