Crypto Bluebook

Version 1.2.3

data
4life

Contents

List of figures

List of tables

Part I
Personal Health Data Platform

1 Introduction

Data4Life develops and operates a data platform called Personal Health Data Platform (PHDP) that uses end-to-end encryption (E2EE) to allow users to securely store and access healthcare data, receive healthcare data from external sources (like hospitals), and selectively share data with third parties (like doctors or other healthcare professionals). This document describes the cryptographic protocols[1] that implement the above-mentioned data-related tasks.

1.1 End-to-end encryption

Before users can access our platform, they must first register and validate an account with us. This establishes a private and end-to-end encrypted data storage that can be accessed by the applications we offer. By an *application* we refer to either a browser-based application or a native mobile app. In most cases we will collectively refer to them as *client applications*, or just *clients*. We will often refer to the private data storage as just the *server*.

Users may, for example, select documents from hard disk and upload them into their data storage. The documents can later be downloaded again. At no point in time does Data4Life get access to the unencrypted data. Data is always encrypted on the client before it is transmitted to the server. Similarly, when downloading documents, the encrypted data is sent from the server to the client where it is decrypted and displayed.

In addition to manually uploading documents, users can also grant one or more third parties append-only access to their data storage. This way, a hospital may transmit discharge letters or similar documents directly into the users' data storage. Apart from granting append-only access once, no further user interaction is necessary.[2]

[1] We adopt the notion of a cryptographic protocol from [10] and consider them to *"consist of an exchange of messages between participants."*

[2] And, of course, users can revoke that write-only access at any time.

1.2 Document structure

Chapter 2 introduces cryptographic concepts required to follow the subsequent chapters:

- Chapters 3 and 4 describe the internal data model and the protocols for user data upload and user data download.
- Chapter 5 explains how third parties can securely write into users' data storage.

2 Cryptographic basics

We assume the reader has enjoyed some elementary exposure to the topics of cryptography. This document is no attempt to provide a thorough treatment of the theoretical and practical underpinnings of the vast field that is cryptography. We will briefly revisit the topics that are required to follow the next chapters. For further reading we refer to the standard literature [2, 10, 17, 14, 20].

2.1 Symmetric encryption

Encryption is the principal goal of cryptography; it makes data incomprehensible in order to ensure its confidentiality [2]. The data or documents to be encrypted are also often referred to as *messages* or *plaintexts*. An algorithm called a *cipher* gets as input the plaintext and a secret (the *key*) and produces the encrypted output called the *ciphertext*.

If the same key is used for encryption and decryption, we are dealing with *symmetric encryption* and a *symmetric cipher*. Figure 2.1 illustrates this scenario. As it is customary in cryptographic literature, we use special given names when denoting the participating parties in a cryptographic protocol. As usual, Alice wants to send a secret message to Bob across an insecure channel which might be eavesdropped on by some malicious party Eve.

Both Alice and Bob share the secret key k. Alice uses the encryption function Enc of some symmetric cipher to encrypt message m with key k to produce the ciphertext C which gets sent across the insecure channel to Bob who uses the same key k and the decryption function Dec of the symmetric cipher to reproduce the message m from the ciphertext C.

Data4Life uses the Advanced Encryption Standard cipher with 256-bit keys (AES-256) for all symmetric encryption.

2.1.1 Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES) is a symmetric block cipher which became a standard in 2001 [8] after it emerged as the lead candidate in a U.S. government call-for-algorithms to replace the outdated DES cipher [16]. The block size of AES is 128 bits, that is, the encryption and decryption functions of it operate on 128-bit

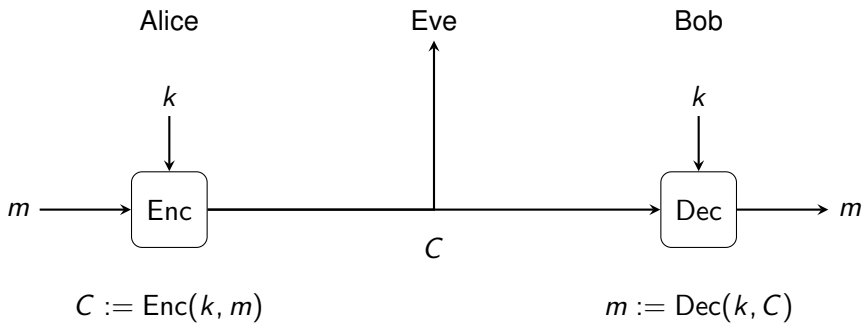

Figure 2.1: Symmetric encryption and decryption between Alice and Bob.

messages and 128-bit ciphertexts, respectively. AES supports three key lengths: 128, 192 and 256 bits. At Data4Life we exclusively use 256-bit keys and refer to it as AES-256. Hence, the Enc function (and also Dec) of Figure 2.1 has the following signature for AES-256:

$$\mathsf{Enc}_{\mathsf{AES}-256} : \{0,1\}^{256} \times \{0,1\}^{128} \to \{0,1\}^{128}$$

Block ciphers such as AES encrypt only a fixed number of bits at a time. For longer messages (and in our case a message will be a healthcare document, possibly as large as an X-ray image) we need to iteratively apply the cipher on consecutive chunks of the message in order to fully encrypt it (and likewise for decryption). There are various ways to iteratively apply a block cipher, which is known as the *modes of operation*. Modes of operations are encryption algorithms of their own, but they need a block cipher as a "plugin" to carry out the actual encryption. Note also, that a mode of operation is not necessarily tied to a specific block cipher. We will discuss some of them below in the context of AES-256, of course, but any other block cipher would do.

2.1.2 Electronic Code Book Mode (ECB)

A straightforward, but problematic approach to extend AES, or any block cipher, to arbitrary message sizes is to split up the original message m into n chunks of size 128 bits each[1] and encrypt those chunks m_i individually:

$$\begin{aligned} C_i &:= \mathsf{Enc}(k, m_i), \quad \text{for} \quad i = 1, \ldots, n \\ m_i &:= \mathsf{Dec}(k, C_i), \quad \text{for} \quad i = 1, \ldots, n \end{aligned}$$

While this idea has some desirable properties (encryption and decryption can be parallelized trivially, and random access to the ciphertext is also possible),[2] it maps

[1] We will discuss further down how to handle the case when the message length is not an integer multiple of the block size, that is, when the last block is not full.

[2] Assume the message m is an ultrasound video and one wants to fast-forward to, say, the middle. One does not have to decrypt the first half of the ciphertext to reach the middle, but one can directly seek to the center block and start decrypting from there.

same message blocks to same ciphertext blocks, that is:

$$m_i = m_j \quad \Leftrightarrow \quad C_i = C_j$$

As a result of this block independence, each repetition of a plaintext block results in repetition of the corresponding ciphertext block, presenting a cryptographic weakness by unnecessarily revealing structural information about the message [10]. This mode of operation is not used in any Data4Life application for its obvious security compromise. It is included here only to motivate the need for different modes of operation.

2.1.3 Cipher Block Chaining Mode (CBC)

The main idea to overcome the drawback of ECB mode is to make a ciphertext block C_i not only dependent on its message block m_i and the key k, but also on the previous ciphertext block C_{i-1} and thus indirectly dependent on all previous ciphertext blocks:

$$\begin{aligned} C_i &:= \mathrm{Enc}(k, m_i \oplus C_{i-1}), \quad \text{for} \quad i = 1, \ldots, n \\ m_i &:= \mathrm{Dec}(k, C_i) \ \oplus \ C_{i-1}, \quad \text{for} \quad i = 1, \ldots, n \end{aligned}$$

Before encryption, each message block m_i is combined with the previous ciphertext block C_{i-1} using the XOR ($\oplus$) operation. This effectively randomizes the message block using the previous ciphertext block. The only question remaining is what value to pick for C_0, which is needed to produce the first ciphertext block C_1. This value is called an *initialization vector* and there are various strategies to pick it [10].

At Data4Life we use AES-256 in CBC mode with zero initialization vector to encrypt and decrypt so-called *tags*. The zero initialization vector is used to achieve a deterministic ciphertext. See Chapter 4 for details.

2.1.4 Padding

The two modes discussed so far essentially "keep the block nature" of the underlying block cipher. That is, a message is still encrypted and decrypted block-wise using the block cipher's encryption and decryption function, respectively. In this scenario it can happen that the last chunk m_n of a message does not have full block size. For example, a 100-byte message would—assuming AES—result in seven blocks which all have to be 128 bits in length in order to work for AES. However, the last block would only contain 32 bits of data, leaving 96 bits unused ($800 = 6 \cdot 128 + 32$).

Some *padding* has to be applied to fill up such last blocks without compromising the security of the cipher and also to enable the receiving party to detect those additional bits and remove them in order not to confuse them with message data [10]. As we will see below, there are modes of operation that do not require padding.

2.1.5 Output Feedback Mode (OFM)

Even though Output Feedback Mode is not used at Data4Life, we use it here to motivate certain properties of GCM (which we describe in the next section), which is used to encrypt all data (except tags). OFM does not use the block cipher to directly encrypt the message. Instead it uses the encryption function Enc of the block cipher to produce a pseudorandom key stream k_i. The actual encryption is implemented by XORing the message bits (interpreted as a message stream) with the corresponding bits of the key stream. Decryption is carried out by XORing the ciphertext bits with the same key stream bits.

$$C_i := m_i \oplus k_i \qquad k_i := \mathrm{Enc}(k, k_{i-1}), \quad \text{for} \quad i = 1, \ldots, n$$
$$m_i := C_i \oplus k_i \qquad k_0 := \mathrm{IV}$$

Since the XOR operation is carried out on bit level, no padding is required. This essentially constructs a *stream cipher* out of a block cipher. Also note that the decryption function of the underlying block cipher is never used. The receiving party must just recreate the exact same key stream to decrypt the message. Hence, the same mechanism for creating the key stream must be carried out on both sides.

The initialization vector IV must be unique for each message and must never be repeated because it determines the key stream (together with the key k, or course). The initialization vector is typically prepended to the ciphertext stream. It must be unique, but it is not kept secret.

2.1.6 Galois/Counter Mode (GCM)

Using Galois/Counter Mode [7] constructs a stream cipher from the underlying block cipher. It also has another feature which the modes discussed above did not exhibit: *authenticated encryption*. All encryption ciphers and modes discussed so far catered for one thing only: message confidentiality. However, there is no means to detect errors or malicious modifications in the ciphertexts. That is, if a bit flipped in the ciphertext, the receiver Bob would still get a result from the decryption. Only this result would be different from the original plaintext that Alice sent. However, it would be up to Bob to detect whether the decrypted message has changed from its original. From the cipher and mode algorithm point of view, everything worked out fine. After all, the XOR operation and the internal workings of AES operate on bit level and do not care about any higher-level format.

For the receiver of a ciphertext it is desirable to learn from the decryption algorithm whether every bit has made it through the communication channel unaltered. This is what authenticated encryption caters to. The output of GCM will not only contain the ciphertext, but also a so-called *authentication tag* which acts as a cryptographic checksum that can be used to detect modifications. To be more precise, GCM offers *authenticated encryption with associated data (AEAD)*. Alice, in addition to the

secret message m, may also add another piece of information A, which does not get encrypted, but is covered by the authentication tag.

The intricate workings of GCM go beyond the scope of this document, but the following pseudo-code shows the "GCM API" from a programmer's point of view:[3]

Encryption:
- Input: - message m
  - key k
  - additional data A
  - initialization vector IV
- Output: - ciphertext C
  - additional data A
  - authentication tag T (covering IV, C and A)

Decryption:
- Input: - ciphertext C
  - key k
  - additional data A
  - initialization vector IV
  - authentication tag T
- Output: - message m, or error

If Alice wants to send a message m and additional data A to Bob, a random initialization vector IV is chosen and the concatenation ($\|$) of the following data gets transmitted to Bob. (The additional data A, if specified, would be encoded into C and is not explicitly mentioned.)

$$\text{IV} \,\|\, C \,\|\, T$$

Again, A is protected by the authentication tag, but not encrypted. At Data4Life we do not use the additional data A, but only the authentication tag protection of the ciphertext and initialization vector.

Bob can split the received data into parts IV, C and T. This is possible because the lengths of the initialization vector and the authentication tag are parameters of the underlying protocol and must be negotiated beforehand with Alice. Bob can then use the decryption of GCM to validate the authentication tag T. If that fails, the decryption must be aborted altogether because the communication channel must be considered compromised. Else, he can read the additional data A and decrypt the message m.

2.2 Asymmetric encryption

The main problem to solve with symmetric encryption is key distribution. How does Bob get hold of key k in Figure 2.1? Alice and Bob must either meet in person or use a secure communications channel. However, if there already is a secure communication channel, then why not transmit the secret message m through it in the first place? In addition, if n people want to communicate with each other (still, assuming confidential one-to-one communication), then $\frac{n^2-n}{2}$ keys must be exchanged, that is, the number of keys in the system grows quadratically.

[3] Adopted from [3].

In asymmetric encryption (or, equivalently, public-key encryption) the key for encryption is different from the key for decryption. Each user creates a *key pair* of two mathematically linked keys. Data encrypted with one key can only be decrypted using the other key. One key of the pair will be called the *public key* and can be distributed freely to anybody who wants to send a message. The other key is the *private key* which must never be disclosed to the public. Figure 2.2 illustrates this process. Alice wants to send an encrypted message to Bob. She asks for Bob's public key E_{Bob} and uses it to encrypt the message. The ciphertext is sent to Bob who can decrypt it using his private key d_{Bob}.

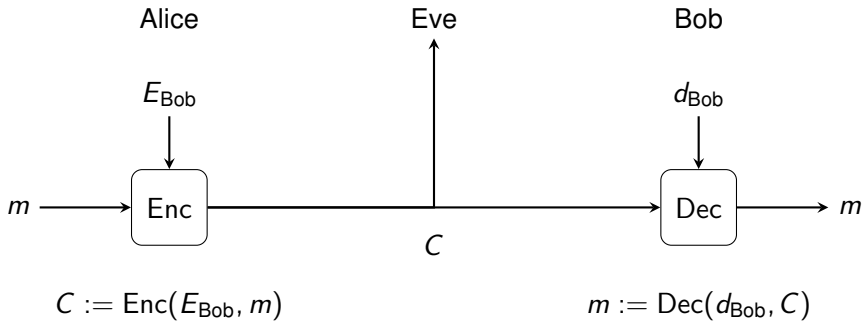

Figure 2.2: Asymmetric encryption and decryption between Alice and Bob.

That is, the following equality holds for all messages m:

$$\text{Dec}(d_{\text{Bob}}, \text{Enc}(E_{\text{Bob}}, m)) = m$$

This approach greatly simplifies the key exchange problem. For a group of n people, only n public keys need to be shared.

So far we only discussed the mathematical link within an asymmetric key pair that guarantees that a ciphertext encrypted with one key can only be decrypted with the other. But we haven't discussed how to construct an asymmetric cipher around that. Without going into detail (which is out of scope for this document), the main idea is to use so-called *trapdoor functions* to construct an asymmetric cipher. The value of a trapdoor function can be easily computed, however, the inverse is hard to impossible to compute without a certain secret piece of information.

The most widely used implementation of such a public-key encryption scheme is RSA [19] which is based on large integer factorization. To derive the private key from a public key, an attacker would have to find the prime factorization of a very large integer N which is the product of two very large prime numbers p and q. There are other asymmetric ciphers which draw their security from other hard mathematical problems like, for example, Elgamal [9] which is based on the discrete logarithm problem. At Data4Life we use the RSA-OAEP encryption scheme [4].

2.3 Hybrid encryption

Symmetric and asymmetric ciphers have certain advantages and disadvantages, some of them being complementary to both [14]. Some properties in which both cipher types differ significantly are data throughput, key sizes and key management effort. Symmetric ciphers are orders of magnitude faster during encryption and decryption. Also, key sizes are typically smaller. Finally, asymmetric encryption schemes shine when it comes to key management. Because the private key isn't shared, the number of public keys to be shared is much smaller compared to a symmetric key distribution,[4] and asymmetric key pairs typically have a longer lifetime compared to symmetric keys.

Therefore, if Alice wants to send some large message m to Bob, in practice the hybrid encryption protocol depicted in Figure 2.3 is used: Alice randomly generates a symmetric *session key* k, which she will only use once for the next communication with Bob. Alice encrypts the message m using this session key k, thus producing the ciphertext C_m which can safely be sent to Bob. To allow Bob to decrypt the message, he also needs to securely receive k. This is achieved by asymmetrically encrypting it using Bob's public key, thus producing ciphertext C_k. Bob can decrypt k using his private key and then decrypt the message.

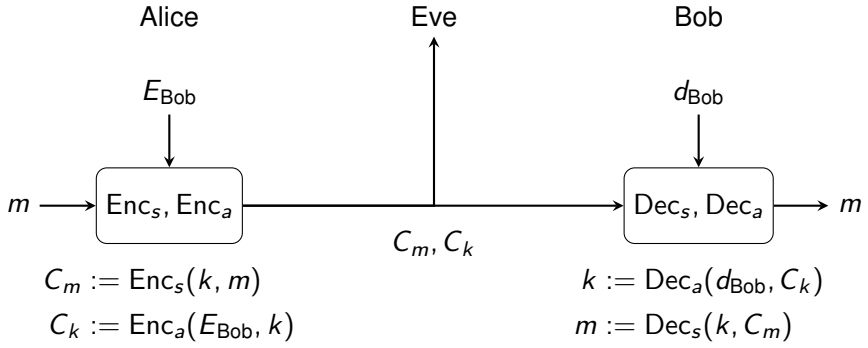

Figure 2.3: Hybrid encryption and decryption between Alice and Bob.

[4] If n persons want to communicate one-to-one, then $\frac{1}{2}(n^2 - n)$ symmetric keys need to be exchanged. Using asymmetric encryption, only n public keys need to be distributed.

3 Secure keys

All healthcare data at Data4Life is end-to-end encrypted. That means, any healthcare data is encrypted on the client before the ciphertexts are sent to the user's data storage. Likewise, when transferring data out of the data storage, ciphertexts are sent to the client where they are decrypted. Furthermore, any communication between client and server is protected via TLS 1.2 [5] currently. TLS 1.3 is planned for the future.

In this chapter we will cover the different cryptographic keys used in our protocols. We use hybrid encryption to encrypt the healthcare data (that is, documents, images and the like) as described in Section 2.3. So far we exclusively considered *healthcare data* or *healthcare documents*. We stick to this simplification for describing the encryption mechanisms we use. In Chapter 4 we will introduce the actual finer-grained data model. However, all concepts discussed below apply there as well.

3.1 Keys overview

When a user first registers her account with Data4Life, she chooses a password w_P which must comply with certain security requirements.[1] Each time at login, after the user was successfully authenticated, the client derives from the password w_P a symmetric key k_P^{F}. See Section 3.2 below for a precise description of the key derivation process.

Each document is symmetrically encrypted with its own *data key* which is generated by the client when the document is first uploaded. Data keys must be stored alongside their document ciphertext in order to be accessible by multiple clients, for example a browser and a mobile app. Therefore, each data key is symmetrically encrypted using a *common key*. The first such common key[2] is generated during account creation. A single common key typically protects multiple data keys. Common keys may be shared with a third-party to allow access to specific documents.[3] On access

[1] It must be at least eight characters total length, contain mixed case letters, contain at least one numeric character (0–9) and at least one special character (such as %, @, or #).

[2] There might be multiple common keys. This is discussed in Chapter 4.

[3] Whoever has access to a common key can decrypt all documents whose data keys are encrypted using this common key, provided this party also has access to the document ciphertexts. The server uses business logic to restrict access to only those document ciphertexts that the third party actually

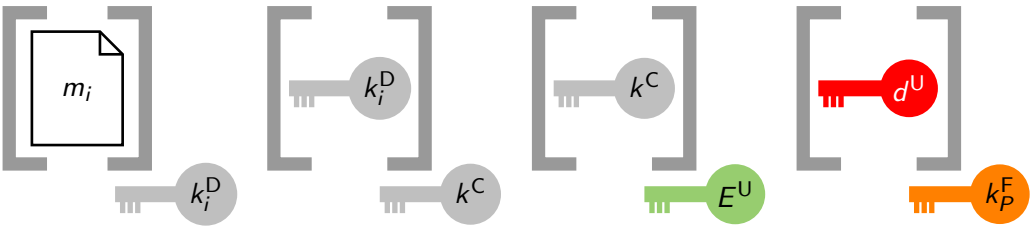

Figure 3.1: Key encryption relationships.

revocation, only the common key needs to be rotated. This entails re-encrypting on the client side all data keys the former common key protects. Common key rotation is a much more efficient process than re-encrypting entire documents with fresh data keys.

Being part of the encryption chain, the common key must also be stored on the server. It is asymmetrically encrypted using the *user public key* which is the public key of a key pair that is also generated during account creation. Lastly, the *user private key* of that key pair is symmetrically encrypted using the password-derived key k_P^F and stored on the server as well.

Let us revisit the key dependencies: Figure 3.1 illustrates the encrypted keys that are stored on the server. We introduce the following short-hand notation to talk more concisely about the various keys in use:[4] Symmetric keys are denoted by a lowercase k, private keys by a lowercase d and public keys by an uppercase E. The key type is annotated in the superscript: D for data key, C for common key, U for user key, F for a functionally-derived key. In Figure 3.1, gray brackets with an offset key mean that the object in brackets is encrypted using the offset key at the lower right. The document m_i is symmetrically encrypted using its data key k_i^D. The data key is symmetrically encrypted using the common key k^C, which in turn is asymmetrically encrypted with the user public key E^U. The corresponding user private key d^U, finally, is symmetrically encrypted using the password-derived key k_P^F. The key k_P^F is not stored on the server, but derived in the client each time the user logs in with the password w_P.

Let us practice notation and key relationships yet again, but this time from the client's point of view. That is, which steps are to be performed to display a document m_i in the client? Figure 3.2 illustrates this without visual clutter by using the notation established above. We borrow from Figure 3.1 the convention to abbreviate $\mathrm{Enc}(k, m)$ by $\left[m\right]_k$. Note how the four ciphertexts in the first row of Figure 3.2 coincide with the ciphertexts depicted in Figure 3.1.

Ultimately, we want to access the plaintext of document m_i. Its ciphertext $\left[m_i\right]_{k_i^D}$ is read from the server.[5] In order to decrypt the document ciphertext, we need

was authorized to see.

[4] Chapter 6 will introduce the full notation.

[5] We will discuss authorization checks later in Chapter 4. Obviously, the server is not handing out

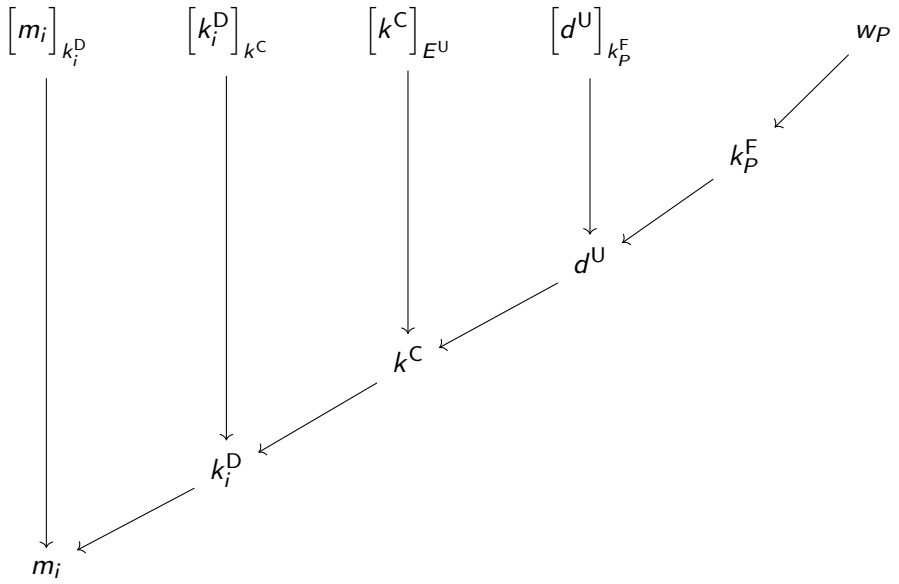

Figure 3.2: Keys and key ciphertexts involved to decrypt document m_i.

Key type	**Protected/encrypted by**
Data key	Common key
Common key	User public key
User private key	Password-derived key
Password	Remembered by user

Table 3.1: Protection and encryption relationships of the different key types.

the corresponding data key k_i^D whose ciphertext $\left[k_i^D\right]_{k^C}$ is read from the server, typically along with $\left[m_i\right]_{k_i^D}$. To get access to the data key k_i^D, we need the common key k^C. Its ciphertext $\left[k^C\right]_{E^U}$ is also retrieved from the server, and it requires the user private key d^U to be decrypted. The ciphertext $\left[d^U\right]_{k_P^F}$ of the user private key is finally decrypted using the password-derived key k_P^F, which exists in the client only ephemerally. The diagonal sequence in Figure 3.2 represents the chain of keys required to decrypt a document. Table 3.1 summarizes the relationships between the key types.

ciphertexts to unauthorized parties.

3.2 Account creation

We demonstrated earlier in this chapter that the encryption of a document and its data key requires a couple of other keys to be present in the system, namely the common key, and the public and private user keys.[6] Those keys are generated during account creation (or, synonymously, account registration).

The *user key pair*, consisting of the user private key and user public key, remains unchanged during a user account's lifetime. All other keys can be changed by the user or are rotated during certain operations. When registering a new account, the user is asked for her e-mail address and a password. A hash of the password is sent to the *haveibeenpwned* API [1] to check whether it might be compromised. If the check passes, the following keys and salts are randomly sampled by the client:

User key pair This is an RSA key pair of length 2048 bits. We denote it by (d^U, E^U) and refer to d^U as the *user private key* and E^U as the *user public key*. This key pair does not change throughout the lifetime of the user account.

Recovery password In case the user forgets her password, it can be reset using this recovery password. It is essentially a second password w_R which is the BIP-39 mnemonic [15] of a random 128 bit number. The user has only one opportunity to download this recovery password, as a PDF file, during the account creation. The respective screen is shown in Figure 3.3. If the user loses this password and forgets her own chosen password, she cannot access the account anymore and all data becomes undecryptable. An example recovery key might look like this:

> grunt runway wet horror tent economy
> garment photo pause dice achieve soul

Common key The common key consists of 256 bits of randomness that make up an AES key. So far we have denoted it as k^C, but we will add an index like in k_0^C for the remainder of this document, because there may be multiple common keys present in the system simultaneously. The reasons for this are covered in Chapters 4 and 5. However, after account creation, there is initially just a single common key.

Tag encryption key As we will see in Chapter 4, each document can have associated metadata called *tags*. A tag might contain the document type or other, non-healthcare data. We symmetrically encrypt those tags with the *tag encryption key* k^T. This key also remains constant throughout the lifetime of an account.

[6] In reality there is one more key involved: the tag encryption key. It encrypts metadata of the document and we omit it here for brevity. It is explained in Chapter 4.

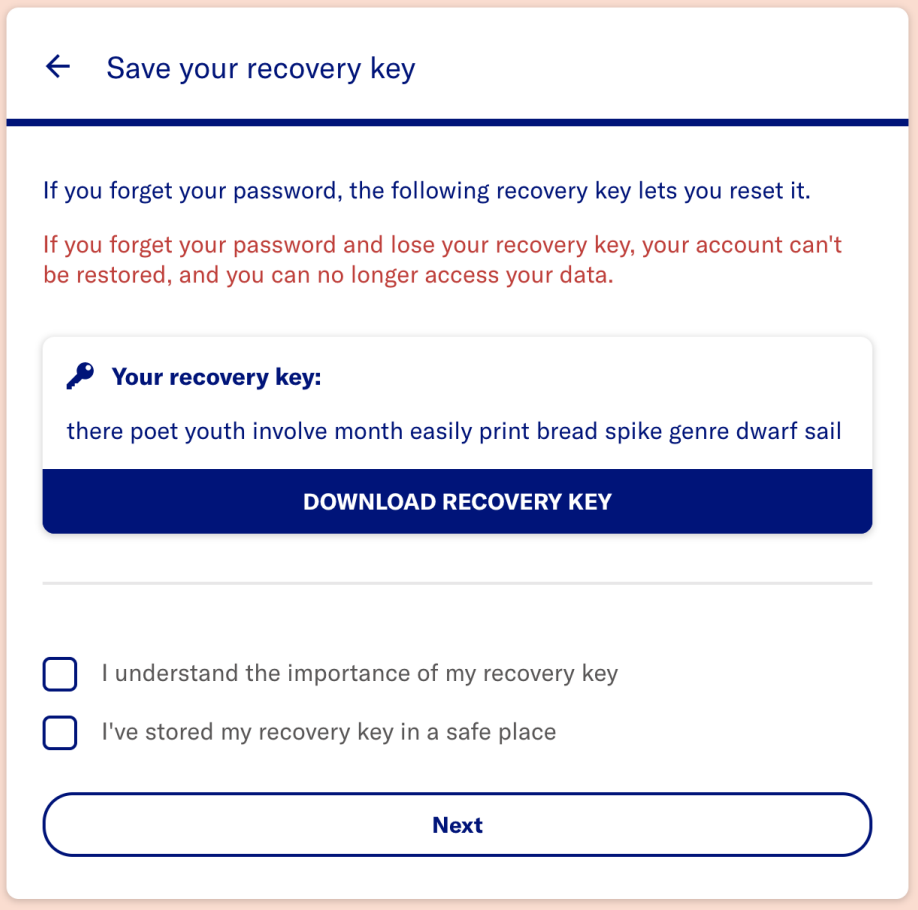

Figure 3.3: The user has exactly one opportunity to download the recovery key during account creation.

Data element	Encrypted with
User e-mail address	
User password derivative hash	
Recovery password derivative hash	
User password salt	
Recovery password salt	
User public key	
Encrypted user private key	User-password-derived key
Encrypted user private key	Recovery-password-derived key
Encrypted common key	User public key
Encrypted tag encryption key	Common key

Table 3.2: Data elements sent to the server when creating a new user account.

User password salt and recovery password salt Two salt values with 16 bits of randomness each are generated for the user and the recovery password. The user salt is used to derive the symmetric key k_P^F. The recovery salt is used for a similar reason, in case the user password is to be reset (see Section 3.4 below).

User password derivative and recovery password derivative The client does never send raw passwords to the server during account creation or login, but rather a key derivative of it. We use the output of the PBKDF2 [13] key derivation function (10,000 iterations with zero salt and SHA-256) as a derivative value.

Clicking "Next" on the screen of Figure 3.3 will send the data given in Table 3.2 to the server where it makes up the user's account. Before password derivatives are sent to the server for storage, they are hashed using *bcrypt* [18] with a cost parameter of 10. The resulting hashes are encrypted at rest with a secret server key using AES in GCM.

3.3 Login

Apart from authentication and authorization, the login procedure serves another important purpose called *client approval*. This approval establishes fresh keys for encrypting and decrypting documents. Any data-related server API endpoint requires a valid JSON Web Token (JWT [12]) to be contained in client requests. Login and client approval, if successful, will result in such a JWT.

3.3.1 Authentication and authorization

To log in, the user is asked for her e-mail address and password. The e-mail address and the password derivative (see Table 3.2) are sent to the server. If the e-mail is known, the hash of the password derivative matches the one stored on the server,

and a second factor (in our case a PIN received via text message on the phone) was provided, the user is considered authenticated. The login request will also contain a list of so-called *scopes*, which are strings designating specific access rights.[7] If the scopes are allowed for the user and the requesting application (that is, the application that triggered the login), authorization succeeds, and client approval is commenced.

3.3.2 Client approval

As described above, the user key pair is a long-lived key that essentially protects all other keys in the system (see Figure 3.1, where it is depicted at the end of the "key chain").[8] Technically, the client could now exercise the flow depicted in Figure 3.2 to decrypt documents of the user, and also to encrypt new documents when proceeding in the opposite direction. However, this requires the user private key to be present in the client for as long as the login session lasts. In order to reduce attack surface, we decided to require every login session to provide its own ephemeral so-called *application key pair*.

Prior to commencing the login process, the client generates a fresh asymmetric application key pair $(d_s^{\mathsf{A}}, E_s^{\mathsf{A}})$. (Let the index s denote the word *session*.) If authentication and authorization succeed, the server sends the ciphertexts of the user private key and of all common keys to the client:

$$\left\{ \left[d^{\mathsf{U}} \right]_{k_P^{\mathsf{F}}}, \left[k_0^{\mathsf{C}} \right]_{E^{\mathsf{U}}}, \left[k_1^{\mathsf{C}} \right]_{E^{\mathsf{U}}}, \ldots \right\}$$

These ciphertexts are decrypted, re-encrypted using the application public key, and sent back to the server which stores them:

$$\left\{ E_s^{\mathsf{A}}, \left[k_0^{\mathsf{C}} \right]_{E_s^{\mathsf{A}}}, \left[k_1^{\mathsf{C}} \right]_{E_s^{\mathsf{A}}}, \ldots \right\} \tag{3.1}$$

The client can decrypt any documents that are currently stored in the user account, regardless which client uploaded them (and, hence, which common keys were used to protect the data keys). This is because it has access to all common keys that are in the user account at the moment. The client approval is now complete. The server will return a JWT that the client must include with all subsequent server calls to prove authenticity and authorization.

3.4 Account recovery

In Section 3.2 we briefly mentioned that the recovery password, which is created and downloadable during account creation, can be used to get access to the account in

[7] It would not be wrong to think of such a scope string as a user role. The notion of a scope is adopted from the OAuth2 authentication protocol [11].

[8] You might argue that the user-password-derived key is the most crucial one—and that is correct—however, the password (hopefully) persists exclusively in the user's memory, and the derived key only ephemerally in the client.

case the user has forgotten her password w_P. Without password w_P, no key k_P^F can be derived, which is required to start the "decryption chain" depicted in Figure 3.2. However, along with $\left[d^U\right]_{k_P^F}$ we also generated $\left[d^U\right]_{k_R^F}$ and stored it on the server during account creation. By choosing a dedicated recovery flow during login and by entering the recovery password w_R, the client can decrypt the user private key d^U, ask the user for a new password $w_{P'}$ and send the new ciphertext $\left[d^U\right]_{k_{P'}^F}$ back to the server.

3.5 Key maps

So far we have discussed a number of key topics. Before we continue, let us establish a common understanding of how the server stores key ciphertexts for each user. Figure 3.4 shows an example. At any given time, a user's data store contains the following individual key ciphertexts:

- User private key encrypted with the user password-derived key: $\left[d^U\right]_{k_P^F}$
- User private key encrypted with the recovery password-derived key: $\left[d^U\right]_{k_R^F}$
- Tag encryption key encrypted with the current common key k_x^C: $\left[k^T\right]_{k_x^C}$

Figure 3.4 depicts these three ciphertexts in the first row (the current common key is k_1^C). Further, each user account contains one or more key maps (or, more precisely, common key maps). A key map contains for each asymmetric key pair (user key pair or application key pairs) the corresponding public key and the ciphertexts of all relevant common keys encrypted using the respective public key.[9] One common key ciphertext in each key map is marked as the key map's *current common key*. We denote this by a little dot above the corresponding ciphertext in Figure 3.4. Finally, a key map may also contain the ciphertext of the tag encryption key.

There is at least one key map $\pi_\emptyset$ present in a user account. It holds the user public key and ciphertexts of all the common keys stored in the system for the user. When a user logs in, the login session that is established, results in a new key map, like π_s in Figure 3.4. The contents of that key map are the result of the client approval (see Equation 3.1 above). For example, π_s contains the session public key E_s^A and the ciphertexts of the three common keys k_0^C, k_1^C and k_h^C that were present at the time of client approval. Key map π_h was not created during a login client approval, but during the so-called onboarding of a third party h (see Chapter 5 for details). For now it is sufficient to accept that the key maps may contain ciphertexts of several common keys. There is exactly one dot above one common key in each map, denoting the current common key for the respective session or third party.

[9] Relevant means that not every key map contains the ciphertexts of all common keys.

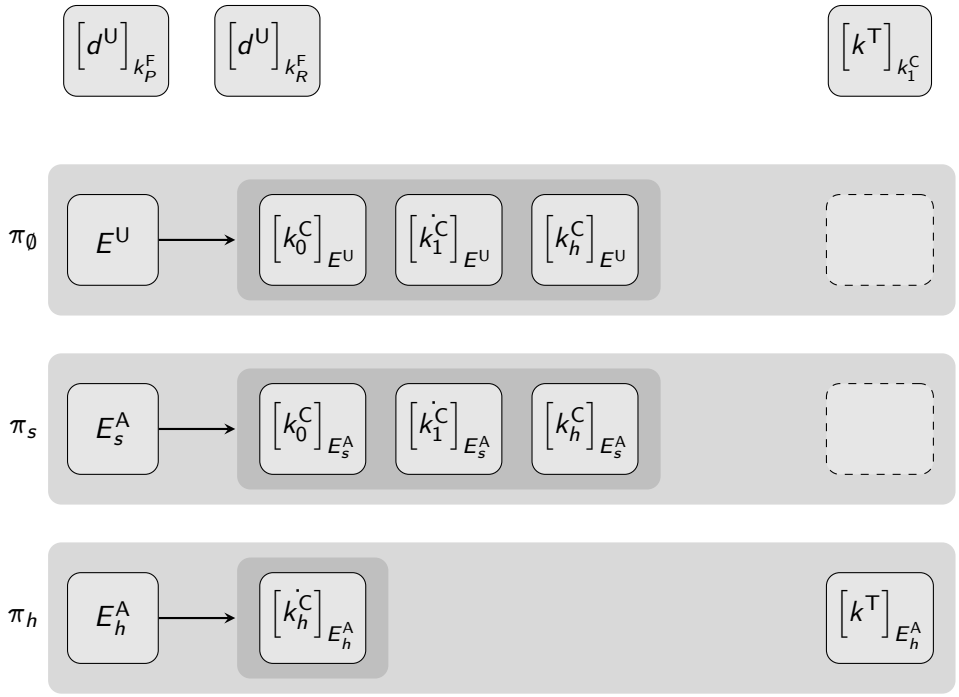

Figure 3.4: Example of a user account with three key maps containing ciphertexts of different common keys. Key map $\pi_\emptyset$ is present in every account and maps the user public key to the ciphertexts of all common keys known to the account. Any other key map my contain a subset of them. Key map π_s contains the ciphertexts of the same three common keys, while key map π_h contains only the ciphertext of common key k_h^C. π_h also contains the ciphertext of the tag encryption key.

4 Data model and data access

This chapter describes the full data model of the data storage together with the cryptographic protocols used to write and read data by a user. We discussed the main ideas in Chapter 3, but we left out some details covered here.

So far we talked about documents that users can maintain in their data storage, and that is certainly a metaphor worth adhering to. The actual data model is more granular, though. There is the notion of a *record*, which can be considered a folder that contains so-called *attachments*. When the user selects one or more files from disk to be uploaded, a new record is created and the files are stored as associated attachments. It is not a mistake to consider the unit of a record and its attachments as a document in the sense we used it so far. Each record can have zero ore more associated *tags*. Tags are strings (typically key-value pairs) which can contain operational data such as a data type. The record itself has a body which is a FHIR resource [6] and as such essentially just a JSON string.

Figure 4.1 illustrates the relationships. The record r_i be a JSON FHIR resource of type `DocumentReference`. The three attachments $a_{i,1}$, $a_{i,2}$ and $a_{i,3}$ could be PDF files, for example. The two tags $t_{i,1}$ and $t_{i,2}$ be some metadata (for example, tag $t_{i,1}$ could be `uploadedVia=mobile`). The attachment IDs, here, exemplary, 1 and 2, but in reality these would be random UUIDs, are stored inside the record's FHIR body.

The record body is symmetrically encrypted using its corresponding data key k_i^{D}, which is used for this record only and nowhere else. In case a record is updated, a new data key is generated and used.[1] Each record is associated with a another symmetric key, the *attachment key* k_i^{N}, which is used to encrypt all its attachments. Each tag is symmetrically encrypted using the account-wide *tag encryption key* k^{T}. As shown in Table 3.1 in Chapter 3 (page 16), the data key (and also the attachment key) are encrypted using the common key, or, more precisely, the *current* common key k_x^{C}. For technical reasons we tolerate multiple common keys to be in the system. However, at any given time there is only one marked as current key and used for encrypting new data uploaded by the user. Table 4.1 summarizes the data that is stored on the server for our example record r_i.

Record and attachments are encrypted using AES-256 in GCM.[2] The initialization

[1] Essentially, a record update is deletion followed by creation of a new record.
[2] See Section 2.1.6 for details.

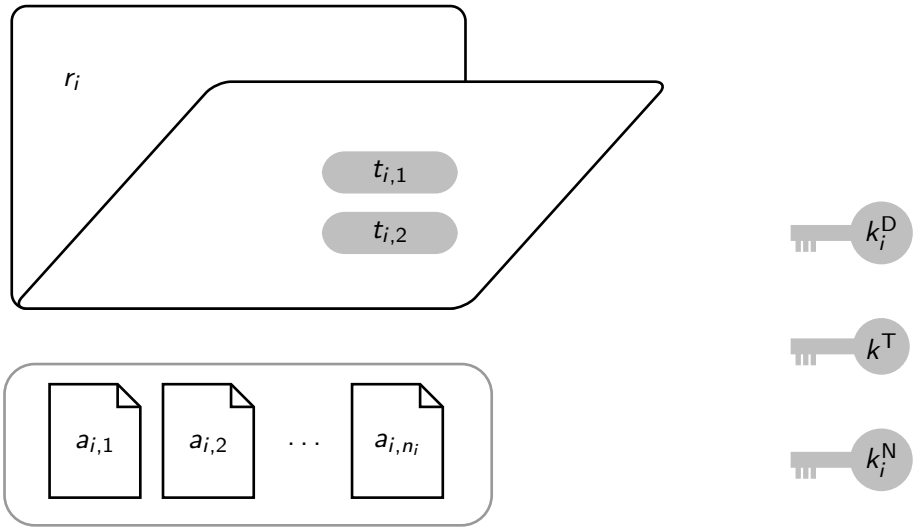

Figure 4.1: Data model example.

Data element	**Formal notation**
Encrypted record body	$\left[r_i\right]_{k_i^{\mathsf{D}}}$
Encrypted data key	$\left[k_i^{\mathsf{D}}\right]_{k_x^{\mathsf{C}}}$
Encrypted attachment key	$\left[k_i^{\mathsf{N}}\right]_{k_x^{\mathsf{C}}}$
Encrypted tags	$\left[t_{i,1}\right]_{k^{\mathsf{T}}}, \left[t_{i,2}\right]_{k^{\mathsf{T}}}$
Encrypted attachments	$\left[a_{i,1}\right]_{k_i^{\mathsf{N}}}, \left[a_{i,2}\right]_{k_i^{\mathsf{N}}}, \left[a_{i,3}\right]_{k_i^{\mathsf{N}}}$
Common key ID	x

Table 4.1: Data elements stored for a record r_i with two tags and three attachments.

vector consists of 12 bytes of randomness and the authentication tag is 16 bytes long. The binary payload that gets stored for an encrypted record or encrypted attachment consists of the concatenation of the initialization vector, the actual ciphertext, and the authentication tag as depicted in Figure 4.2. Tags are symmetrically encrypted using AES-256 in CBC mode with the initialization vector consisting of 16 zero bytes.

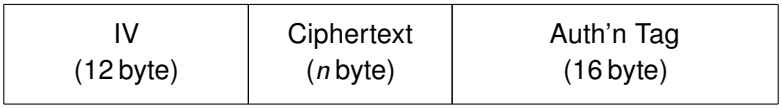

Figure 4.2: Data layout for symmetrically encrypted records and attachments.

Let us now revisit which steps are taken when a user wants to display the attachment $a_{i,1}$ of record r_i. Since the attachment ID is stored in the record body, it must be decrypted first. This is shown in Figure 4.3 (this chart is essentially identical to Figure 3.2 up to the namings of some artifacts, like message m_i versus record r_i and the like). After decrypting the record r_i, the corresponding attachment IDs are known. This allows to retrieve the ciphertext $\left[a_{i,1}\right]_{k_i^N}$ and proceed analogously for decrypting it as depicted in Figure 4.4.

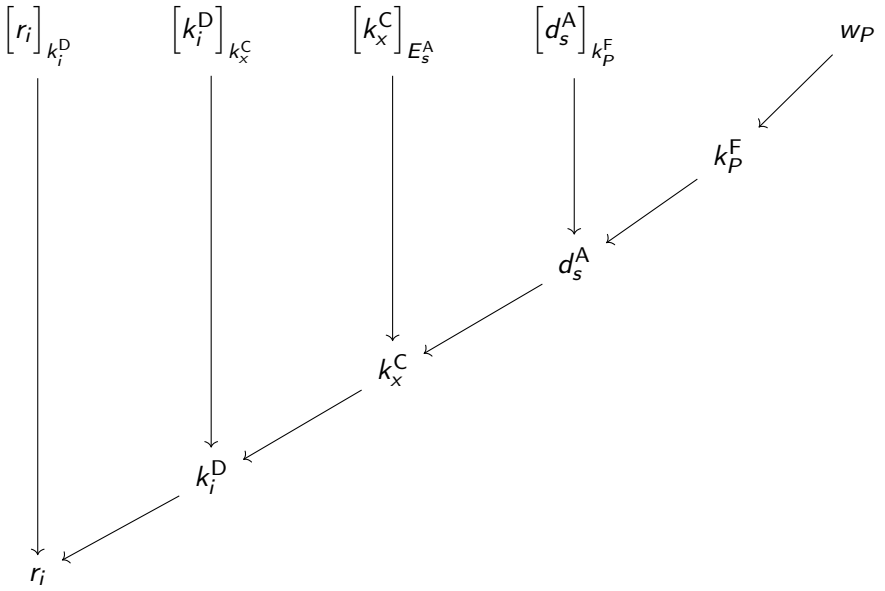

Figure 4.3: Data decryption flow for reading record r_i.

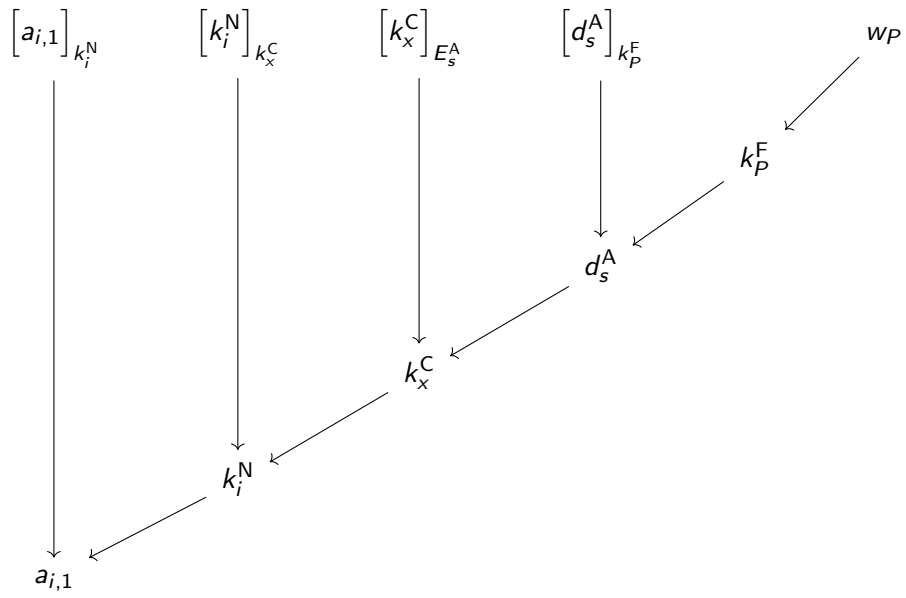

Figure 4.4: Data decryption flow for reading attachment $a_{i,1}$ after having read the record r_i.

5 Data ingestion

One way of adding documents to a user's data storage is to use the web application or mobile application and upload data manually. This was covered in Chapter 4. We provide another avenue by which healthcare documents generated by hospitals (such as doctor letters or discharge letters) can be transferred securely directly into the user's data storage.

The main challenge is to enable a third party, for example a hospital, to securely add documents to a user's data storage while not allowing them any read access. As we saw in Chapter 3, a common key protects data keys, that is, data keys are encrypted by a common key. Any party who holds a common key can decrypt all documents[1] whose data keys are encrypted with it. Note, that this argument implies that a third party would have access to all ciphertexts of those documents, which, in practice, will not be the case, of course.

The main idea for enabling a third party to add documents is to generate an additional common key which is securely transferred to the third party for encrypting the data keys of the new documents. The upload of encrypted documents by a third party is cryptographically identical to the upload of a document by the user, except that the common key used for data encryption keys is different. Table 5.1 illustrates this subtle but important difference. When a user uploads a document, its data key is encrypted using the current common key (here assumed to be still the initial one k_0^C). When a hospital h uploads a document for the user, the data key is encrypted using the hospital's dedicated common key k_h^C. Note that the hospital would hold such a dedicated common key for each user who granted them write access. That is, we should more precisely denote the hospital's common key of user u as ${}^{(u)}k_h^C$ to stress that user binding. In the remainder, however, we assume the user to be implicitly fixed and omit the (u) decoration.

[1] For the sake of simplicity, we again talk about documents, but do, of course, recall, that they consist of records and attachments (see Chapter 4). Nevertheless, all arguments and ideas put forth w. r. t. documents in this chapter carry over to records and attachments.

Party	Common key	Ciphertexts sent for document m_j
User	k_0^C	$\left[m_j\right]_{k_j^D}$, $\left[k_j^D\right]_{k_0^C}$, $\left[t_{j,*}\right]_{k^T}$
Hospital h	k_h^C	$\left[m_j\right]_{k_j^D}$, $\left[k_j^D\right]_{k_h^C}$, $\left[t_{j,*}\right]_{k^T}$

Table 5.1: Ciphertexts sent to the server by user vs. a third party such as a hospital.

5.1 Onboarding

Before a third party can add documents to a user account, it has to undergo onboarding in a special type of login process. The third party h generates an application key pair (d_h^A, E_h^A) and sends the public key E_h^A to the authentication and login application at Data4Life.[2] The application detects that we are in an onboarding flow instead of a regular login flow, and performs the following steps:

1. Generate a new common key k_h^C for third party h.
2. Encrypt it using the provided public key: $\left[k_h^C\right]_{E_h^A}$
3. Execute a modified client approval:
   - Add to each non-foreign key map π_x in the user account the ciphertext of the new common key: $\left[k_h^C\right]_{E_x^A}$. *Non-foreign* refers to key maps which do not hold keys for another third party. In other words, do not add the ciphertext to any key map which was established using this very protocol.
   - Add the new key map $\pi_h = \left(E_h^A, \left\{\left[k_h^C\right]_{E_h^A}\right\}\right)$ to the user account.
   - Add to the key map π_h the tag encryption key ciphertext $\left[k^T\right]_{E_h^A}$.
4. Allow third party h to read the contents of π_h, that is, the ciphertexts $\left[k_h^C\right]_{E_h^A}$ and $\left[k^T\right]_{E_h^A}$, at any later point in time by issuing a JWT that allows requests against the server. See Section 3.3 for details.[3]

This concludes the onboarding process of a third party.

[2] The very same web application that implements the account creation, login, client approval and account recovery flows.

[3] In fact, we are issuing two tokens: a short-lived JWT for immediate use and a long-lived refresh token, which can be used at a later point in time to request a new JWT (this is identical to the principles of OAuth2 [11]). Since the third party will rarely add documents immediately after onboarding, the JWT will likely expire. It is the refresh token that is kept by the third party and exchanged for a fresh JWT prior to the actual document upload.

5.2 Document upload

Adding a document by the third party works as follows:

1. Exchange refresh token for a JWT to authenticate subsequent server requests.
2. Request the ciphertexts of π_h from the server: $\left[k_h^C\right]_{E_h^A}$ and $\left[k^T\right]_{E_h^A}$
3. Decrypt k_h^C and k^T using the private key d_h^A
4. Encrypt a document m_j as usual, that is, produce: $\left[m_j\right]_{k_j^D}$ and $\left[k_j^D\right]_{k_h^C}$
5. Encrypt the necessary tags, say, $t_{j,1}$ and $t_{j,2}$ with the tag encryption key k^T.
6. Upload ciphertexts $\left[m_j\right]_{k_j^D}$, $\left[k_j^D\right]_{k_h^C}$, $\left[t_{j,1}\right]_{k^T}$ and $\left[t_{j,2}\right]_{k^T}$ to the server.

Even though the third party did add a document, there is no way for them to decrypt any document ciphertext other than the ones they uploaded. Assume the ciphertext of some document m_x leaks. The corresponding data key k_x^D is encrypted with a common key different from k_h^C and thus the third party cannot decrypt it. There is no way for party h to decrypt any other common key. Its key map does not include them and other key maps, should the party get hold of them via an attack, are encrypted with public keys other than E_h^A.

5.3 Document access by client

Let us verify that the user can read the uploaded documents using any approved client (that is, any application for which there exists a key map in the account). We can, in fact, reuse Figure 3.4 on page 22 to illustrate this. The key map π_h was created during the onboarding of third party h. Let the key map π_s belong to a session that the user just logged in to. Assume further, that party h has uploaded a doctor's letter m_j. In order to decrypt it in the client (session s), the following steps are performed:

1. Download ciphertexts $\left[m_j\right]_{k_j^D}$ and $\left[k_j^D\right]_{k_h^C}$.
2. Key map π_s contains $\left[k_h^C\right]_{E_s^A}$. Decrypt it using d_s^A to get k_h^C.
3. Use k_h^C to get k_j^D, which then is used to get m_j.
4. Tags $t_{j,1}$ and $t_{j,2}$ can be decrypted, because k^T is decryptable via k_1^C which is in key map π_s.

The client, which can only access its session key map π_s, can decrypt the data key because the required common key k_h^C was inserted into π_s during approval of the third party onboarding.

Part II
Formal treatment

6 Cryptographic notation

This chapter introduces short-hand notations that we use to accurately, yet concisely, define cryptographic protocols. We already implicitly used some notation in Chapter 2 and introduced some more in Section 3.1.

The main objectives for the formal notation are short names for cryptographic symbols (such as keys, messages or tags) and the encryption itself. Further, the notation should work at a whiteboard, on paper and have flavor that can be used in plain ASCII (for example in source code, Markdown or Slack). We use the following syntax to denote a cryptographic symbol or data element:

$$ {}^{(u)}s_b^{\mathrm{A}} $$

It is a main symbol (here: s) which can be decorated with one index and up to two superscripts.

Symbol s

The main symbol is a single character that represents the type of object as listed in Table 6.1. A lowercase symbol indicates that the object is protection-worthy and must not be disclosed to the public. An uppercase symbol denotes an object which may (or must) be made public for certain protocols to work.[1]

Superscript A

The superscript is used to indicate a subtype when used for certain symbols, like keys or passwords. See Table 6.2 for details.

Index b

We use the subscript for indexing or enumeration. The subscript itself can be complex or comma-separated or both. See the examples in Table 6.3.

[1] This was inspired by the Go programming language where lowercase symbols are private while uppercase symbols get exported.

Symbol	Meaning
k	symmetric key
d	private key
E	public key
r	record
a	attachment
t	tag or token (clear from context)
S	salt value
w	password
m, v	other confidential message or value
M, V	other non-confidential message or value

Table 6.1: Different types of cryptographic symbols.

Symbol	Meaning
U	user-related
A	application-related
C	common key
D	data key
N	attachment key
T	tag-related
F	function-derived (e. g. via a key derivation function)
S	service-related

Table 6.2: Superscripts indicating subtypes of cryptographic symbols.

Superscript (u)

We occasionally use this second superscript for tagging the symbol with a user ID u, if different users shall be differentiated explicitly. The superscript is written in parentheses.

Encryption

Typically, when denoting the encryption of some data x using a key k, some notation of the following is used:

$$\mathrm{Enc}(k, x) \quad \text{or} \quad \mathrm{Enc}_k(x) \quad \text{or} \quad E_k(x)$$

This notation can take up quite some screen real estate if many ciphertexts are part of the game (and this is the case for our platform). That is why we denote the ciphertext of x using key k as follows:

$$\left[x\right]_k$$

The precise cipher being used (symmetric or asymmetric) will be clear given the key notation.

ASCII flavor

We want to use the notation also for source code comments as well as in Slack communication. For this purpose we use the following ASCII rendering:

$$^{(u)}s_b^{\mathrm{A}} \quad \Leftrightarrow \quad \texttt{(u)sA\_b}$$

The main instances of the notation are listed in the following Table 6.3.

Symbol	ASCII	Meaning
d^{U}	`dU`	User private key (of implicit user)
$^{(u_1)}d^{\mathrm{U}}$	`(u1)dU` or `(u_1)dU`	User private key of user u_1
E^{U}	`EU`	User public key (of implicit user)
k_0^{C}	`kC_0`	Initial common key (of implicit user)
k^{T}	`kT`	Tag encryption key (of implicit user)
$\left(d_1^{\mathrm{A}}, E_1^{\mathrm{A}}\right)$	`(dA_1, EA_1)`	Application key pair of app ID 1
$\left(d_s^{\mathrm{A}}, E_s^{\mathrm{A}}\right)$	`(dA_s, EA_s)`	Application key pair for web app session s
w_P	`w_P`	User-chosen password
w_R	`w_R`	Recovery password
k_P^{F}	`kF_P`	Key derived from user password w_P
k_R^{F}	`kF_R`	Key derived from recovery password w_R

Symbol	ASCII	Meaning
$\left[d^{U}\right]_{k_P^F}$	`[dU]kF_P`	User private key encrypted using password-derived key
r_i	`r_i`	Record i
k_i^D	`kD_i`	Data key of record r_i
k_i^N	`kN_i`	Attachment key of record r_i
$\left[r_i\right]_{k_i^D}$	`[r_i]kD_i`	Ciphertext of record r_i encrypted using data key k_i^D
$\left[k_i^D\right]_{k_0^C}$	`[kD_i]kC_0`	Ciphertext of data key k_i^D encrypted using initial common key k_0^C
$a_{i,j}$	`a_i,j` or `a_(i,j)`	j-th attachment of record r_i
$\left[a_{i,3}\right]_{k_i^N}$	`[a_i,3]kN_i`	Ciphertext of 3rd attachment of record r_i
$t_{i,j}$	`t_i,j` or `t_(i,j)`	j-th tag of record r_i
$\left[t_{i,j}\right]_{k^T}$	`[t_i,j]kT`	Ciphertext of j-th tag $t_{i,j}$ (of record r_i) using tag encryption key k^T

Table 6.3: Examples of the cryptographic notation.

7 Account creation

When a new user registers an account with Data4Life, the following steps are carried out (see also Section 3.2 for a shorter but less precise description).

Client side

In the client application, the user enters the values shown in Table 7.1. A hash of the entered password is checked against the *haveibeenpwned* API [1] and in case of a positive result the user must choose another password. The same check is carried out when users want to change their password. The client application then computes and generates the data listed in Table 7.2. The data that gets sent to the server is listed in Table 7.3.

Server side

Most of the user registration payload becomes a row in the `users` table. The password hashes H_P and H_R are hashed again using *bcrypt* [18] to form H_P^* and H_R^*, respectively. The user public key and the encrypted initial common key become the zero permission

$$\pi_\emptyset := \left({}^{(u)}E^{\mathsf{U}},\ \left\{ \left[{}^{(u)}k_0^{\mathsf{C}} \right]_{{}^{(u)}E^{\mathsf{U}}} \right\} \right).$$

See Figure 7.1 for a graph depicting the data dependencies of the user registration payload.

Material	Description
${}^{(u)}V_{\text{email}}$	User u enters e-mail address
${}^{(u)}w_P$	User u enters password

Table 7.1: Data required to be entered by the user when registering a new account.

Material	Description
${}^{(u)}S_P$	User password salt
${}^{(u)}k_P^{\mathsf{F}}$	Derive key ${}^{(u)}k_P^{\mathsf{F}} := \mathrm{PBKDF2}\left({}^{(u)}w_P, {}^{(u)}S_P, N\right)$ with salt ${}^{(u)}S_P$ and N iterations
${}^{(u)}w_R$	Generate recovery password (BIP-39 mnemonic)
${}^{(u)}S_R$	Recovery password salt
${}^{(u)}k_R^{\mathsf{F}}$	Derive key ${}^{(u)}k_R^{\mathsf{F}} := \mathrm{PBKDF2}\left({}^{(u)}w_R, {}^{(u)}S_R, N\right)$ with salt ${}^{(u)}S_R$ and N iterations
$\left({}^{(u)}d^{\mathsf{U}}, {}^{(u)}E^{\mathsf{U}}\right)$	Generate user key pair
${}^{(u)}k_0^{\mathsf{C}}$	Generate initial common key
${}^{(u)}k^{\mathsf{T}}$	Generate tag encryption key

Table 7.2: Automatically generated data by the client.

Material	Description
${}^{(u)}V_{\mathsf{email}}$	User e-mail address
H_P	User password hash $H_P := \mathrm{PBKDF2}\left({}^{(u)}w_P, 0, N\right)$
H_R	Recovery password hash $H_R := \mathrm{PBKDF2}\left({}^{(u)}w_R, 0, N\right)$
${}^{(u)}E^{\mathsf{U}}$	User public key
$\left[{}^{(u)}d^{\mathsf{U}}\right]_{{}^{(u)}k_P^{\mathsf{F}}}$	User-password-encrypted user private key
$\left[{}^{(u)}d^{\mathsf{U}}\right]_{{}^{(u)}k_R^{\mathsf{F}}}$	Recovery-password-encrypted user private key
${}^{(u)}S_P$	User password salt
${}^{(u)}S_R$	Recovery password salt
$\left[{}^{(u)}k_0^{\mathsf{C}}\right]_{{}^{(u)}E^{\mathsf{U}}}$	Encrypted common key
$\left[{}^{(u)}k^{\mathsf{T}}\right]_{{}^{(u)}k_0^{\mathsf{C}}}$	Encrypted tag encryption key

Table 7.3: User registration payload that gets sent to the server.

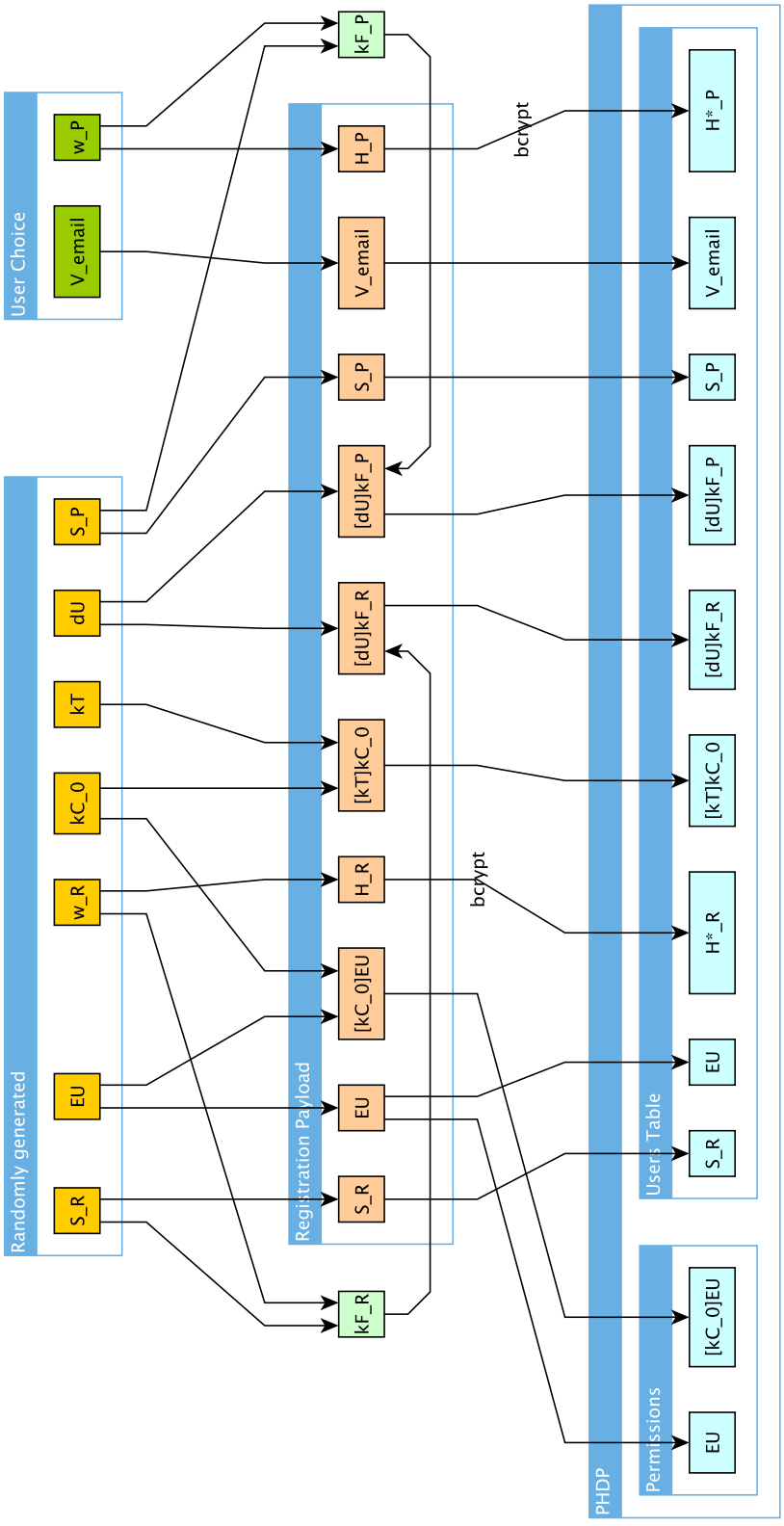

Figure 7.1: Dependencies of data elements needed for user account creation.

Part III
Appendices

A Bibliography

[1] *';–have i been pwned?* URL: https://haveibeenpwned.com/ (visited on 07/30/2020) (cit. on pp. 17, 35).

[2] Jean-Philippe Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. USA: No Starch Press, 2017. ISBN: 978-1-59327-826-7 (cit. on p. 7).

[3] *Authenticated encryption*. Wikipedia. URL: https://en.wikipedia.org/wiki/Authenticated_encryption (visited on 05/05/2020) (cit. on p. 11).

[4] Mihir Bellare and Phillip Rogaway. "Optimal Asymmetric Encryption – How to Encrypt with RSA". In: Springer-Verlag, 1995, pp. 92–111 (cit. on pp. 12, 41).

[5] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. RFC Editor, Aug. 2008. URL: http://www.rfc-editor.org/rfc/rfc5246.txt (cit. on p. 14).

[6] *Documentation FHIR v4.0.1*. HL7 FHIR Website. URL: https://www.hl7.org/fhir/documentation.html (visited on 05/13/2020) (cit. on p. 23).

[7] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. Tech. rep. SP 800-38D. NIST, 2007. URL: https://doi.org/10.6028/NIST.SP.800-38D (cit. on p. 10).

[8] Morris J. Dworkin et al. *Advanced Encryption Standard (AES)*. Tech. rep. NIST FIPS 197. NIST, 2001. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf (cit. on pp. 7, 41).

[9] T. Elgamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE Transactions on Information Theory* 31.4 (1985), pp. 469–472 (cit. on p. 12).

[10] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, Inc., 2010. ISBN: 978-0-470-47424-2 (cit. on pp. 5, 7, 9, 42).

[11] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. RFC Editor, Oct. 2012. URL: http://www.rfc-editor.org/rfc/rfc6749.txt (cit. on pp. 20, 28).

[12] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. RFC Editor, May 2015. URL: http://www.rfc-editor.org/rfc/rfc7519.txt (cit. on p. 19).

[13] B. Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. RFC 2898. RFC Editor, Sept. 2000. URL: http://www.rfc-editor.org/rfc/rfc2898.txt (cit. on p. 19).

[14] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. 1st. USA: CRC Press, Inc., 1997. ISBN: 978-0-84-938523-0 (cit. on pp. 7, 13).

[15] *Mnemonic code for generating deterministic keys*. Github.com. URL: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki (visited on 05/07/2020) (cit. on p. 17).

[16] National Institute of Standards and Technology. *Data Encryption Standard (DES)*. Tech. rep. FIPS PUB 46-2. NIST, 1993. URL: https://doi.org/10.6028/NIST.FIPS.46-2 (cit. on pp. 7, 41).

[17] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st. Springer Publishing Company, Inc., 2009. ISBN: 978-3-642-04100-6 (cit. on p. 7).

[18] Niels Provos and David Mazières. “A Future-Adaptive Password Scheme”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’99. Monterey, California: USENIX Association, 1999, p. 32 (cit. on pp. 19, 35).

[19] R. L. Rivest, A. Shamir, and L. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342 (cit. on p. 12).

[20] Bruce Schneier and Phil Sutherland. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 2nd. USA: John Wiley & Sons, Inc., 1995. ISBN: 978-0-471-12845-8 (cit. on p. 7).

B Glossary

Term	Abbrev.	Definition
Advanced Encryption Standard	AES	Symmetric cipher for encryption and decryption of data [8]
authenticated encryption		A cipher which does not only provide confidentiality, but also message authenticity.
authenticated encryption with associated data	AEAD	Authenticated encryption where some additional data can be added which is not encrypted, but covered by the authentication tag.
authentication tag		Cryptographic checksum that guarantees the integrity of a ciphertext.
common key		Symmetric key used to encrypt data keys and attachment keys.
Data Encryption Standard	DES	Outdated symmetric cipher for encryption and decryption of data [16]
end-to-end encryption	E2EE	Encryption and decryption of healthcare data occur at the client. The server never gets in contact with unencrypted (plaintext) healthcare user data.
initialization vector	IV	Random data that is used as the “zeroth” plaintext block to start encryption.
mode of operation		Algorithm to iteratively apply a (fixed-size) block cipher to arbitrary input message lengths.
Optimal asymmetric encryption padding using RSA	RSA-OAEP	Padding scheme used in conjunction with RSA that is proved secure against certain attack types [4].

Term	Abbrev.	Definition
padding		Process of extending a plaintext to give it a length that is a multiple of the supported block size of the used symmetric cipher. Also, it is used with asymmetric encryption to randomize the plaintext in order to avoid certain attack scenarios [10].
Personal Health Data Platform	PHDP	Data storage platform using end-to-end encryption for secure healthcare data
tag encryption key		Symmetric key used to encrypt records tags.
user private key		Private key of an RSA key pair created at user registration and valid throughout the account lifetime.
user public key		Public key of an RSA key pair created at user registration and valid throughout the account lifetime.

C Change history

Version	Date	Remarks
1.0.0	2020-05-13	Initial revision
1.1.0	2020-05-20	Add data sharing, crypto notation and examples
1.1.1	2020-05-25	Adjust wording, fix typo
1.1.2	2020-05-28	Add acknowledgments
1.1.3	2020-06-05	Fix typos
1.1.4	2020-06-06	Adjusted profession wording
1.2.0	2020-06-08	Streamline chapters
1.2.1	2020-06-11	Incorporate feedback
1.2.2	2020-07-08	Fix typo, fix reference
1.2.3	2020-07-30	Add information on password checks

D Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation program under the grant agreement No. 826117: Smart4Health – Building a citizen-centered EU-EHR exchange for personalized health.

DATA4LIFE.CARE

data
4life