Maintaining Confidentiality of Patient Data with Efficient Key Usage on a Shared Database

Sarah Scheffler, Frederick Jansen, Aditi Dass

May 4, 2017

1 Introduction

We have designed and implemented an accessible, lightweight, cryptographically secure framework for storage and access of confidential data by multiple parties. The framework is designed for an organization that has consultants that serve multiple clients. For example, a hospital where each doctor has multiple patients. In this scenario, all doctors and patients can log into the the framework but only the patient and their respective doctor can access the patient's records. A doctor is able to access the records of all the patients they care for. Specialists or other doctors could also be given access to these records if the need ever rises.

In this paper, we proceed by first defining the objectives we intend our system to achieve. The main security and access goals as well a possible threats to this scheme are discussed in section 2. The design and structure of the system are further elaborated on in 3. Finally, the blueprints of the software implementation and an explanation of our choices of algorithms are discussed in 4 followed by possible future work in 5. Our code can be found at `https://github.com/aditid/CryptoProject`.

2 Security Guarantees

Throughout this project, our goal is to implement certain functionality securely even in the presence of adversaries with specific capabilities. We specify what these goals are in section 2.1, and then we discuss the adversaries that we defend against in section 2.2.

2.1 Desired Functionality

The following refer to the range of capabilities that we would like to implement in the system:

Password-based Authentication Similar to many frameworks in use today, a user should be able to authenticate themselves to the server using a username and password. The user should not have to keep track of any information other than their password.

Server Breach Tolerance Client data and records must remain confidential even in the case of a server breach. For example, if a malicious party is somehow able to steal the entire database, they should not be able to read any records. Variations of this are further discussed in section 2.2.

Client Breach Tolerance If a client's login session is compromised, the attacker will have access to all data requested during that session. But they should not be able to access the records for any other user, nor should they be able to decrypt any record they do not have the key to.

Also, as discussed in section 2.2, a compromise of one client login session should not provide an adversary any information that helps them once the login session is over.

Access to Records Records must be accessible by multiple parties, and one party must be able to access multiple records. Records should have confidentiality against parties who are not authorized to read them. Specifically, even the server itself should not be able to read the records unless a client requests access.

Asynchrony If one user has access to a record and wishes to share it with another user, that other user should not have to be online at the same time as the first user in order to gain access. This implies the use of public key encryption, as discussed in section 3.1.

2.2 Threat Model

1. All communication between the client and the server is secure (by Transport Layer Security) and no adversaries will be able to obtain any useful information through passive or active attacks by intercepting network communication.

2. The server is well protected and is able to detect and oppose any long-term passive adversaries. This means any adversary should not be able to lay low and collect private keys over a period of time to eventually decrypt records.

3. The server is not susceptible to a complete breach by any active adversary with the authorization to rewrite code. The responsibility of which lies on the server provider.

4. An adversary cannot corrupt a logged-in client and dump the server state at the same time. Specifically, if a server state dump to the adversary occurs, the adversary cannot corrupt any client with an active login session. The result of this assumption is that the adversary is never able to reconstruct a client's private key. This means that each new login by the client is a "clean slate" in the sense that possessing knowledge about a prior login does not help an adversary in a future login.

We are primarily concerned with malicious clients who either wish to access data of other clients that they should not have access to, or clients whose login sessions have been corrupted by some adversary. We also consider an adversary who is capable of taking a snapshot of the server state at any given time and might dump database information for any or all databases.

3 Design

At its core, our system is meant to encrypt records shared between multiple users. As mentioned in section 2.1, we do not want to require a user to be online for them to gain access to a record. Since we do not want to store a user's decryption key on the server in case of server breach, we must use public key cryptography to accomplish this goal. This is similar to the "protected unless open" class used in the iOS 10 security model [3].

Assuming the records are larger than keys themselves, it makes more sense to assign a symmetric key to each record, encrypt the record under this *record key*, and then wrap the record key under a set of allowed users' public keys. This has several benefits: large encryptions are done with faster symmetric cryptography rather than slower public key schemes, and we must only store one encrypted key per user rather than an entire encrypted record per user.

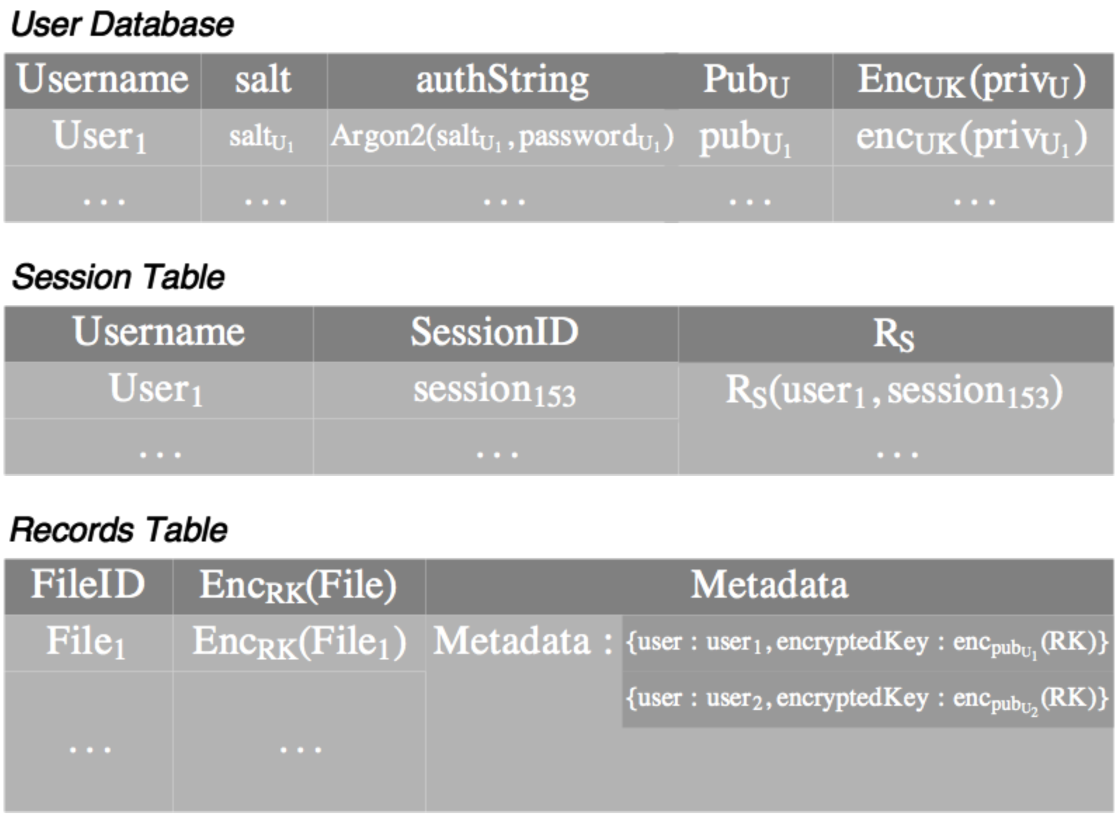
The rest of the system is basically infrastructure to be able to manage these files during user login sessions without sacrificing any functionality described earlier. There are three important databases in this system ; the *User Database*, the *Sessions Table* and the *Records Table*.

3.1 User Database

This database assists in storing any information directly associated with the user. Along with the User's username, the string used for authentication is stored concatenated to a salt. A user's `authString` is an `AES_256` key generated by a user's password using argon2i twice, as described below in section 4.1.

Additionally, each user has a public/private key pair using elliptic curve cryptography with curve 25519, as provided by node-sodium [10]. The public key is stored in the user database under Pub_U. Anyone wishing to grant access to a record simply encrypts the record key under the new user's public key. We use a public key for this (rather than a symmetric key) because this method does not require the user to be online at the time to provide their symmetric key.

The private key, Priv_U, is encrypted with the User Key, or `UK`. Similar to the `authString`, a `UK` is also a `AES_256` key generated by a user's password using argon2i twice but this time with a different salt for the second round.


User Database

Username	salt	authString	Pub_U	$Enc_{UK}(priv_U)$
$User_1$	$salt_{U_1}$	$Argon2(salt_{U_1}, password_{U_1})$	pub_{U_1}	$enc_{UK}(priv_{U_1})$
...	...	...	...	...

Session Table

Username	SessionID	R_S
$User_1$	$session_{153}$	$R_S(user_1, session_{153})$
...	...	...

Records Table

FileID	$Enc_{RK}(File)$	Metadata
$File_1$	$Enc_{RK}(File_1)$	Metadata : $\{user : user_1, encryptedKey : enc_{pub_{U_1}}(RK)\}$ $\{user : user_2, encryptedKey : enc_{pub_{U_2}}(RK)\}$
...	...	

Figure 1: This figure shows the structure of the Database. There are three tables; the User Database, the Records Table and the Sessions Table. respectively.

The encrypted private key, $enc_{UK}(Priv_U)$ is first generated when the user registers for an account. It is never stored even when the user is logged in. Instead, upon logging in and decrypting $Priv_U$, it is split into two "shares" by XOR'ing it with a random value. The random value, R_S is stored in the Session Table (see section 3.2) and the product of the XOR, R_U, is given to the client and stored as a cookie for the duration of a session.

A user can then request a record by sending R_U to the server. By XORing R_S and R_U, the private key can be recreated and used to unlock the record key (see section 3.3). However, each share by itself yields no information about the key. This way, even if a snapshot of the server is taken by a malicious adversary while a user is logged in, the private key cannot be recreated without knowledge of the user's share.

3.2 Session Table

The Session Table keeps track of any and all users who are currently logged in. Upon successfully logging in, an entry is made into the table with the user's username, the session ID and the random number R_S.

The entry lasts for a limited time only and is used to prevent any malicious adversary from accessing files of a breached client beyond the time of the session. This is because the random number R_S is a corresponding component to the users token, R_U, and both must be used together in order to decrypt any of the user's records (see section 3.1).

3.3 Record Table

The Records Table stores the encrypted record as well as the corresponding metadata under the file ID. The Metadata of each record lists the users with access to the record and an encrypted record key for each user.

Each record key is an `AES_256` key used to encrypt a single record. In order to decrypt the record the respective record key, R_K, must first be decrypted. For each user with access to the file, R_K is encrypted under the respective user's public key and can therefore be decrypted with the user's private key. The encryption is authenticated, so the public key of the user performing the initial encryption is also required. This allows the user to verify that the record was created by the right person. As a result of this structure, a user can be given access to a record without having to be online - as R_K can be encrypted under the users public key, Pub_U.

4 Implementation

4.1 Cryptographic Functionality

The main functionality added is the ability to encrypt records. But we want to do this in such a way that we meet the requirements discussed in section 2. Here are descriptions of all the algorithms used.

4.2 Algorithms

This section details the steps followed for doing actions like user login, record encryption and decryption, and so on.

4.2.1 Adding a new user

1. User sends username and password to server. `[username, password]`
2. After checking for username availability, generate a public/private keypair for the user using curve 25519. [Pub_U, Priv_U]
3. Generate a random salt for the user. `[salt]`
4. The server hashes the password with the user's salt using argon2i `[hashedPwd]`, and uses the result for two things:
   (a) Hashing it once more with salt of '0000000000000000' and argon2i producing the authentication string. `[authString]`
   (b) Hashing it again with salt of '0000000000000001' and argon2i to create the User Key. `[UK]`
5. Use the User Key to encrypt the private key, Priv_U, under `AES_GCM`. [$\mathrm{enc}_\mathrm{UK}(\mathrm{Priv}_\mathrm{U})$]
6. Store the salt, authString, Pub_U and $\mathrm{enc}_\mathrm{UK}(\mathrm{Priv}_\mathrm{U})$ under `username` in the User Database.
7. Let User know that registration was successful.

4.2.2 User login

1. User sends username and password to the server. `[username, password]`
2. The server makes a database call to the User Database to retrieve entire object under username. [salt, authString, Pub_U, $\mathrm{enc}_\mathrm{UK}(\mathrm{Priv}_\mathrm{U})$]
3. The server hashes the password with the user's salt using argon2i. `[hashedPwd]`
4. The server then hashes it again with salt '0000000000000000' and argon2i to check for authentication. `[newHash]`
5. If `newHash` matches `authString`, the user is authenticated. Else, an error is returned.
6. `hashedPwd` is hashed with salt '0000000000000001' by argon2i again to produce the User Key. `[UK]`
7. `UK` is used to decrypt $\mathrm{enc}_\mathrm{UK}(\mathrm{Priv}_\mathrm{U})$. [$\mathrm{Priv}_\mathrm{U}$]
8. Server throws away `UK`.

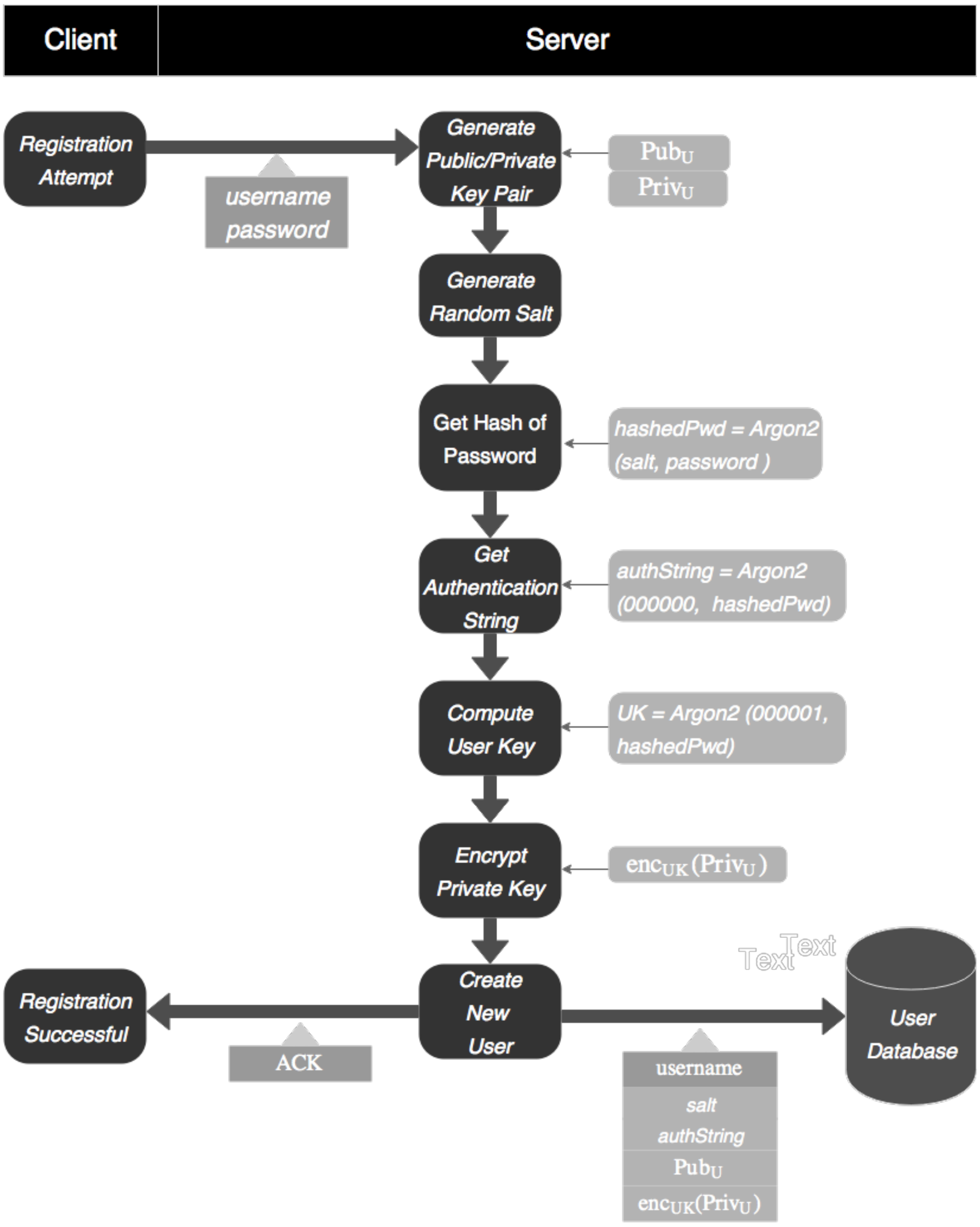


Figure 2: This figure shows the process for a user to register.

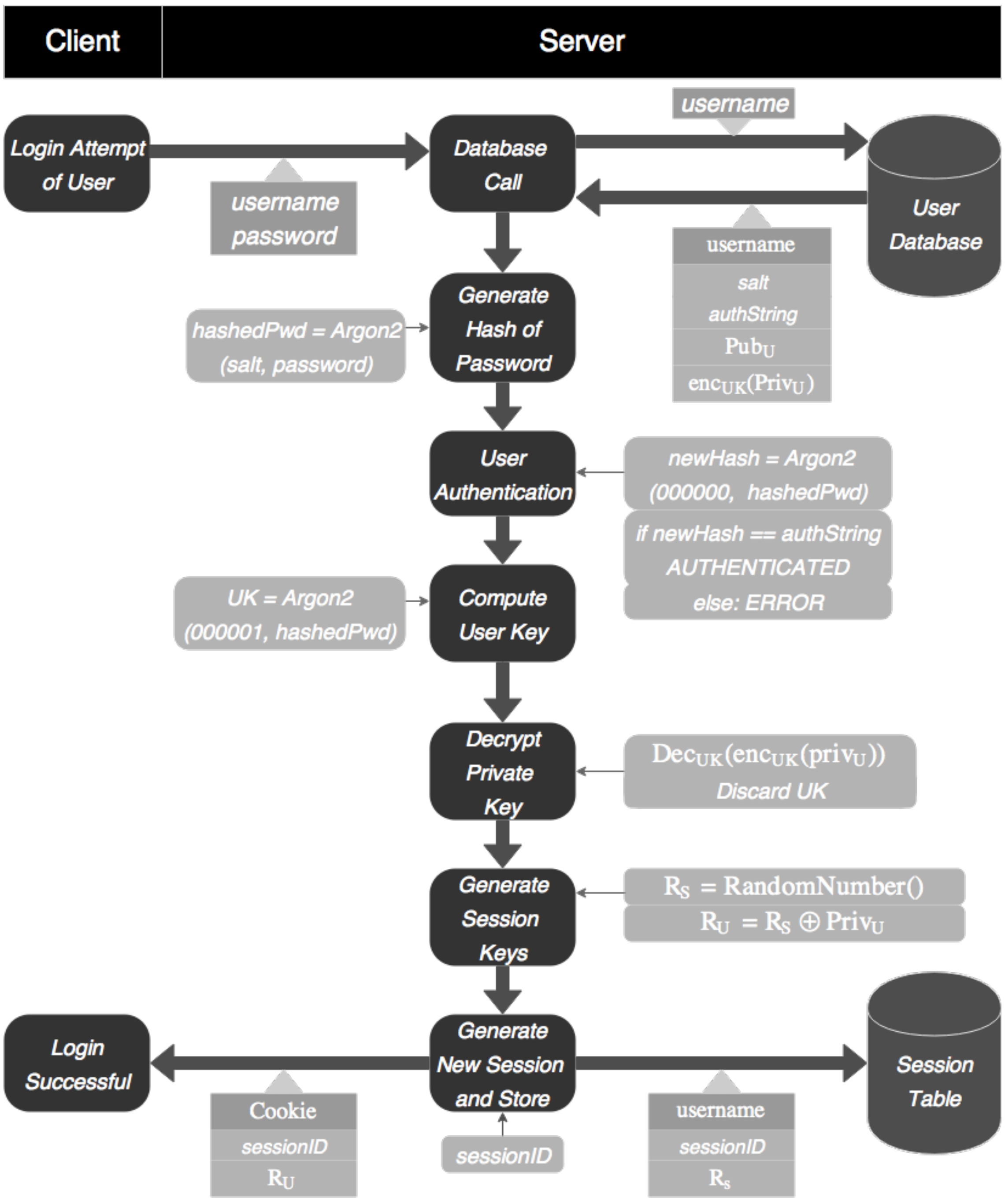


Figure 3: This figure shows the process for a user to log in.

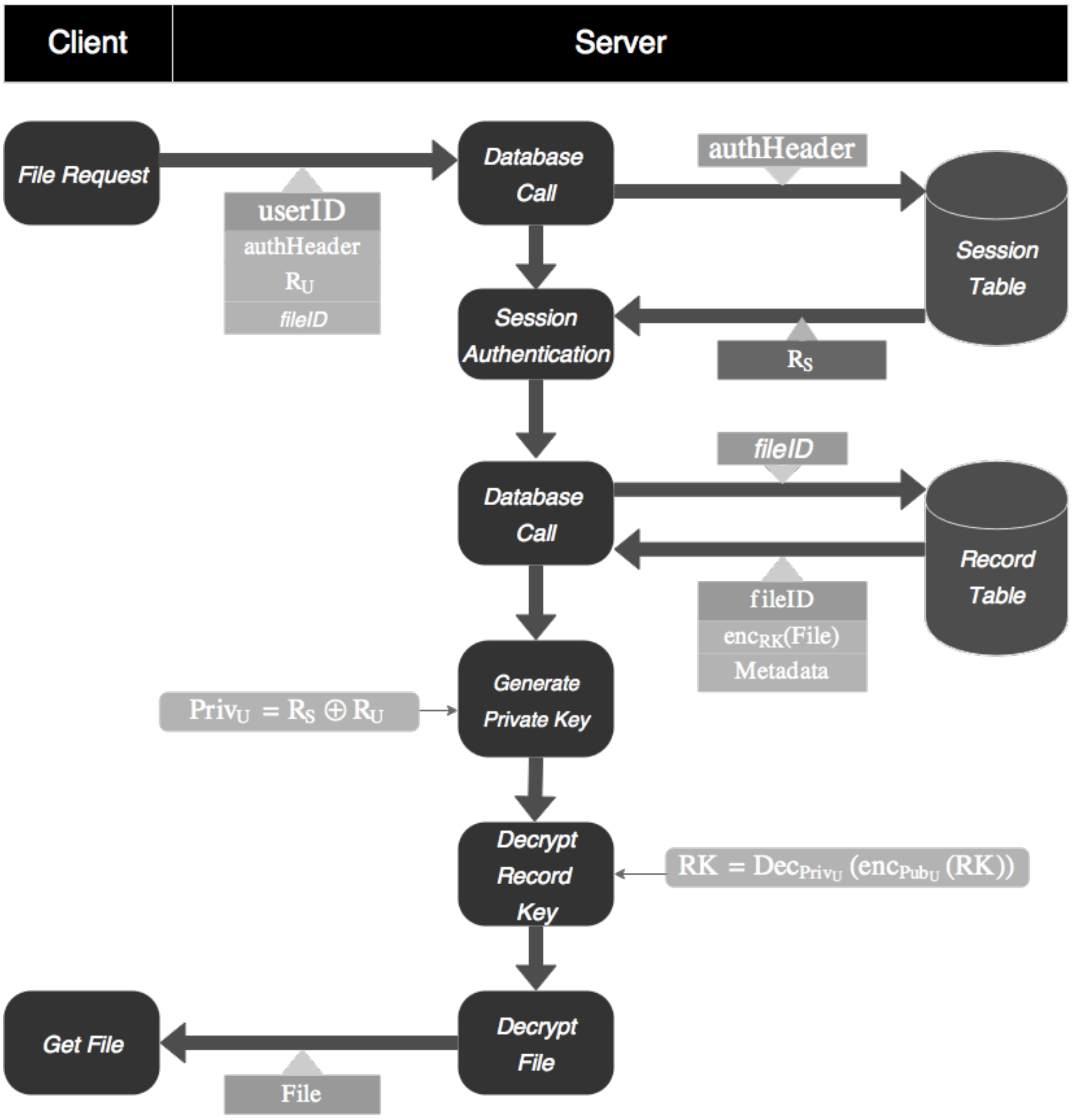


Figure 4: This figure shows the process for a user to request a file.

9. Generate random bits of the same length as the private key to create the server private key share. [R_S]
10. Produce the user private key share by XORing R_S with $Priv_U$. [R_U]
11. Generate unique session number `[sessionID]` and store `sessionID` and R_S under `username` in the Session Table.
12. Send `sessionID` and user private key share (R_U) over to the client where the client will store it as a cookie.

4.2.3 Giving a user permission to read a record (request from a logged-in user)

1. Encrypt the record key under the user's public key and sign with own private key.
2. Store the encrypted key with the other encrypted keys of people who can access this file.

4.2.4 Decrypting a record (request from a logged-in user)

1. User sends the ID of the record they requested as well as the session ID and client private key share (which they had stored as a cookie). [$\mathrm{R_U}$]
2. Server recombines the server and client key shares into the user's private key. [$\mathrm{Priv_U}$]
3. Server calls the database for the file object which includes the encrypted file and metadata. [$\mathrm{enc_{Pub_U}}$, Metadata]
4. Within the file's Metadata, server looks up user and decrypts Record Key. `[RK]`
5. Server decrypts record and sends it to the client.

4.2.5 Creating a new record (request from a logged-in user

1. Generate unique ID for the record.
2. Generate record key (`AES_256`)
3. Encrypt the record key under the user's public key and sign with private key.
4. Optionally: encrypt the record key under an admin's public key.
5. Store the encrypted keys with the record in Metadata; they will be used to decrypt the record.

4.3 Libraries Used

We used two open-source libraries for our implementation. We used Frame[7] for a basic web framework, and node-sodium[10], a JavaScript implementation of the Sodium crypto library[8].

4.3.1 Frame

We chose frame as our web framework both because it was lightweight and because our group already had some familiarity with it. It also proved fairly simple to incorporate our encryption into the existing pages.

4.3.2 node-sodium

Since our entire framework was written in JavaScript, we decided to use a JavaScript implementation of cryptographic code as well. The main Sodium library covers all the functionality we required, and is still maintained and updated regularly. Node-sodium, as a port of Sodium, met our needs and had good documentation describing its use.

4.4 Functions Chosen

We had three major choices to make: a choice of hash function for authentication and key generation, a choice of symmetric encryption scheme for encryption of records and of private keys, and a choice of public key encryption scheme.

4.4.1 Hash Function

We use a hash function to create a string used to authenticate the user, and also to create a separate string used as a symmetric key.

We chose `argon2i` for our hash function. Our set of options was determined by our library: we could use node-sodium's `argon2i` [5] or `scrypt` [11], or alternatively `bcrypt` [12]. Bcrypt is an older function and we wanted to keep our crypto code coming from node-sodium, so the choice was largely between `argon2i` and `scrypt`. We chose argon2i as the winner of the Password Hashing Competition [1] in 2015. It offers improved resistance to side-channel attacks over scrypt[6], and has good documentation describing how to use it for authentication and for key derivation.

4.4.2 Encryption Schemes

We use two encryption schemes in our system: a symmetric key scheme that is used to encrypt records and private keys, and a public key system that is used to encrypt record keys.

Symmetric Key Encryption For symmetric key encryption, node-sodium offers the block cipher `AES_GCM` [9] and the stream cipher `ChaCha` [4]. We chose `AES_GCM` for two reasons. First, because we have the entire bitstring we are encrypting ahead of time; it's not as if we're encrypting a stream of data that we don't know where it will end ahead of time. (This isn't really an argument in favor of `AES_GCM` so much as it is a non-argument for `ChaCha`.) Second, and more importantly, our group was much more familiar with `AES_GCM` and the security guarantees it provides than we were with `ChaCha`.

Public Key Encryption Our choice of public-key cryptography was made for us by our library—node-sodium only provides an elliptic-curve scheme using curve 25519. In fact, even though we weren't necessarily looking for authenticated public key encryption, we got it "for free" in the sense that our only public key option was also authenticated. They use a definition of authenticated public key cryptography detailed by Jee Hea An [2].

5 Future Work

We'd like to add more usability features. Right now we have no ability to deal with a user losing their password, or changing doctors. We think there is potential for the idea of secret-sharing a master key among several trusted parties. We also have a few minor engineering tasks to fix - for example, currently, a user requests all of another user's records that they have access to, rather than requesting a specific record. We should also do a formal proof to demonstrate that our system maintains confidentiality against an attacker that fits our threat model.

References

[1] Password hashing competition. https://password-hashing.net

[2] An, J.H.: Authenticated encryption in the public-key setting: Security notions and analyses. Cryptology ePrint Archive, Report 2001/079 (2001), http://eprint.iacr.org/2001/079

[3] Apple Inc.: iOS Security. https://www.apple.com/business/docs/iOS_Security_Guide.pdf

[4] Bernstein, D.J.: The chacha family of stream ciphers. https://cr.yp.to/chacha.html

[5] Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: the memory-hard function for password hashing and other applications. https://github.com/P-H-C/phc-winner-argon2/raw/master/argon2-specs.pdf (Mar 2017)

[6] Forler, C., Lucks, S., Wenzel, J.: Memory-Demanding Password Scrambling, pp. 289–305. Springer Berlin Heidelberg, Berlin, Heidelberg (2014), http://dx.doi.org/10.1007/978-3-662-45608-8_16

[7] jedireza: Frame: A user system api for node.js. https://jedireza.github.io/frame/

[8] jedisct1: The sodium crypto library. https://download.libsodium.org/doc/

[9] McGrew, D.A., Viega, J.: The Security and Performance of the Galois/Counter Mode (GCM) of Operation, pp. 343–355. Springer Berlin Heidelberg, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/978-3-540-30556-9_27

[10] paixaop: node-sodium: Port of the lib sodium encryption library to node.js. https://github.com/paixaop/node-sodium

[11] Percival, C., Josefsson, S.: The scrypt password-based key derivation function. Tech. rep. (2016)

[12] Provos, N., Mazieres, D.: A future-adaptable password scheme. In: USENIX Annual Technical Conference, FREENIX Track. pp. 81–91 (1999)