HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Département des Technologie de l'information
et de la communication (TIC)
Filière Télécommunications
Orientation Sécurité de l'information

Bachelor Thesis

Password-Authenticated Key Exchange (PAKE)

Author	**Julien Béguin**
Supervisor	Prof. Alexandre Duc
Academic year	2021-2022

Yverdon-les-Bains, January 26, 2022

Specification

Context

Password-authenticated key exchange (PAKE) is a very powerful cryptographic primitive. It allows a server to share a key with a client or to authenticate a client without having to know or to store his password. For this reason, it provides better security guarantees for initializing a secure connection using a password than usual mechanisms where the password is transmitted to the server and then compared to a hash. Despite its theoretical superiority, PAKEs are not implemented enough in the industry. Many old PAKEs were patented or got broken which might have hurt the adoption of this primitive.

Goals

1. Outline existing PAKE. This includes SRP, OPAQUE, KHAPE, EKE, OKE, EKE variants (PAK, PPK, PAK-X,), SNAPI and PEKEP. Also look for other less known PAKEs.
2. Study in detail the main PAKE — EKE, SRP, OPAQUE, KHAPE — and understand their differences.
3. Choose one of the modern PAKEs to implement. The choice is based on the properties of the PAKE, the existence of implementations for this PAKE and the existence of standards for this PAKE.
4. Design an interesting use case where using a PAKE is more appropriate than using a classical authentication method. The advantages of the PAKE are detailed in the report.
5. Implement the chosen PAKE and the use case using the desired programming language

Deliverables

- Implementation of the chosen PAKE with the use case
- Report containing :
  - PAKEs' state of the art,
  - Description of the use case,
  - Advantages of using a PAKE over a classical authentication method for this use case,
  - Implementation details

Contents

1 Introduction

This chapter describes the context of this project. We discuss about classical authentication methods, their weakness and the necessity to use stronger construction such as PAKEs.

1.1 Problematic

How to authenticate a user ? When a user wants to connect itself to an online service, he sends its username or email for identification. Then, he needs a way to prove to the server that he is indeed the person he pretends to be. This is what we call authentication. Without it, anybody can impersonate the account of someone else. Authentication can be based on multiple factors. Something that the user *knows* (e.g., passwords, PINs, . . .), something that the user *has* (e.g., digital certificates, OTP token devices, smartphones, . . .) or something that the user *is* (e.g., fingerprints, iris, . . .). Multiple factors can be combined to obtain a strong authentication. Traditionally, the user sends the authentication value to the server through a secure channel — generally TLS — to avoid eavesdropping and then the server compare the value that he received to the value that he stored for this specific user. This means that the server has to know and store this sensible value before authentication — generally during the register. Currently, the vast majority of web sites and software use passwords as the authentication value. They are the easier to implement and the most familiar to the users.

Attacks and mitigations. This setup is not ideal and can lead to multiple attacks. In case where the server gets compromised, the attacker immediately obtain access to all accounts since the server stores the passwords. This means that the adversary can impersonate every user. To avoid this scenario, numerous techniques have been developed. Mainly, hashing the password and storing the result, adding hashing salt, adding hashing pepper — a secret salt — and using memory-hard password hashing function such as Scrypt [28] or Argon2 [9]. These techniques improve the security of storing password but they do not address the deeper problem; When the user wants to login, he has to send its *cleartext* password to the server in order for the server to authenticate the user. This necessity void any password storing improvement if the server is ever persistently compromised or if passwords are accidentally logged or cached.

Why passwords are bad ? Passwords are a problem. They are hard to remember and to manage for the user. They are generally low-entropy, and users are reusing the same passwords too often. A password manager can help the client to handle this problem, but there is a greater underlying problem. The problem is that "a password that leaves your possession is guaranteed to sacrifice security, no matter its complexity or how hard it may be to guess. Passwords are insecure by their very existence" [11]. Nowadays, a majority of passwords use require that the password be sent in cleartext. Even if the channel between the client and the server is appropriately secured — generally with TLS which can be vulnerable to PKI attack, certificates miss configuration, etc. — and even if on the server side every password storing techniques are carefully implemented, the password still has to be processed in cleartext. As stated before, there can be some software issue like accidental logging or caching of the password. But hardware vulnerabilities are not to forget. While the password is processed in clear, it resides on the memory. It uses a shared bus between the CPU and the memory. Hardware attacks are less likely to occur but are no less severe (remember Spectre [24] and Meltdown [27] attacks). In an ideal world, the server should never see the user's password in cleartext at all. One could think that hashing the password on the client side would solve the problem but if the server ever gets compromised, every account is immediately accessible to the attacker. The client hash should be hashed again on the server, but this does not solve the initial problem. The password is just replaced by a longer password — the hash.

Get rid of password. In summary, passwords are not ideal. They are difficult to remember, annoying to type and insecure. So why don't we try to get rid of them altogether ? Promising initiatives to reduce or remove passwords are emerging and improving — e.g., WebAuthn — but they generally require a deep change for the developers and a sacrifice in convenience for the end user. For example, it can be difficult for an end user to manager private keys if he needs to transfer them securely between multiple devices or if he loses the device that store them. Overall, it will take time for these new solutions to grow mature and impose themselves as industry standard. Passwords are so ubiquitous due in part to the ease of implementation and the familiarity for the users. So if we cannot get rid of passwords for now, we need a way to make them "as secure as possible while they persist" [11]. and this is where PAKE becomes interesting. It allows password-based authentication without the password ever leaving the client.

PAKEs at the rescue. Password-Authenticated Key Exchange (PAKE) is a cryptographic primitive. There are two types of PAKEs:

- Symmetric (also known as balanced) PAKE where the two parties know the password in clear.
- Asymmetric (also known as augmented) PAKE designed for client-server scenarios. Only the client knows the password in clear.

For the moment, we will focus on asymmetric PAKE (aPAKE) because it is the one that can solve our authentication problem. aPAKEs guarantee that the client's password is protected because it **never** leaves the client's machine in cleartext. It is done by computing a key exchange between the two parties and then mutually verifying that they share the same output key. The password is used to compute or retrieve values inputted in the key exchange protocol. This allows a client and a server to mutually authenticate without requiring a secure channel — except for the initial registration. The goal of PAKEs protocol is to provide a shared key between two parties and that the only way for an attacker to execute a dictionary attack — to test a list of password candidates — is to perform online guesses. The attacker is forced to become active and to interact with the server, which is easier to mitigate than passive attacks. Overall, "A secure aPAKE should provide the best possible security for a password protocol" [10] and it should only be vulnerable to inevitable attacks such as online guess or offline dictionary attacks upon server compromise.

Why PAKEs have almost no adoption ? Despite existing for nearly three decades and providing better security guarantees than traditional authentication method, PAKEs have almost no adoption. So why are they so rare in the industry nowadays ? Firstly, for web sites, it is easier to set up a password form and handles all the processing on the server than to implement complex cryptography in the browser. But even in native app PAKEs are rarely used to authenticate. This could be caused by the fact that many old PAKEs were either patented, got broken or both. It probably hurt the reputation and adoption of PAKEs. Another factor is the insufficiency of well-implemented PAKE libraries in some programming language which make them difficult to use. One exception to that is SRP, the most used PAKE protocol in the world [18]. It is a TLS cipher suite is implemented in OpenSSL and used in Apple's iCloud Key Vault. Even though it has far more adoption than other PAKEs, it is not the ideal PAKE. In the last few years, a new generation of strong aPAKE [23, 21] has appeared. These new constructions are better and provide more security guarantees than ever.

1.2 Our contributions

To our knowledge, there is currently no public implementation of KHAPE, so we present the first ever implementation of the KHAPE protocol (Chapters 3 and 4). We also present an implementation of a practical use case using the developed KHAPE library (Chapter 5) and a performance test of the library (Chapter 6). In addition, we summarize the current state of PAKE protocol landscape with a description and comparison between four main PAKEs; EKE, SRP, OPAQUE and KHAPE (Chapter 2).

1.3 Generalities

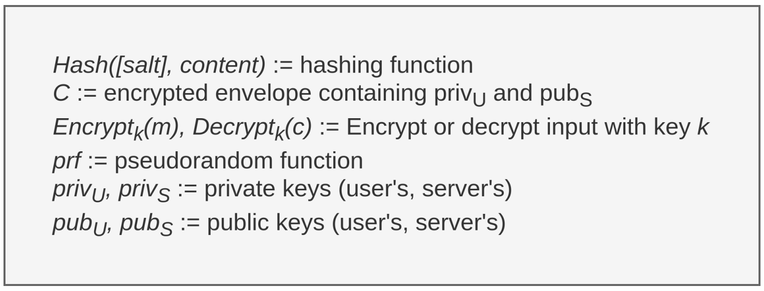
This bachelor thesis concludes a three-year bachelor's degree in information security at HEIG-VD. It started on the 21st of September 2021 and ended on the 24th of December 2021 with an intermediate evaluation on the 26th of October 2021. The expected workload is 450 hours.

2 | State of the art

This chapter aims to provide a detailed view of the current PAKE landscape and its main protocols.

2.1 Notation

Schemas will use the notation described in Fig 2.1.



Hash([salt], content) := hashing function
C := encrypted envelope containing $priv_U$ and pub_S
$Encrypt_k(m)$, $Decrypt_k(c)$:= Encrypt or decrypt input with key *k*
prf := pseudorandom function
$priv_U$, $priv_S$:= private keys (user's, server's)
pub_U, pub_S := public keys (user's, server's)

Figure 2.1: Schema notation.

2.2 Main PAKEs

This section describe in more details four fundamental PAKE construction. EKE as it is the first ever PAKE. SRP because it is the most used. OPAQUE because it is very promising, in the process of standardization and the first construction of this new generation of Strong aPAKE. OPAQUE because it is the first construction of Strong aPAKE KHAPE because it is most recent one and provide slightly better security guarantees than OPAQUE in certain conditions.

2.2.1 EKE

Introduction. EKE (for Encrypted Key Exchange) was proposed in 1992 by Bellovin and Merritt [6] and is the first PAKE protocol. It allows two parties that share a common password to exchange information over an insecure channel. It is a simple protocol that is designed to prevent offline dictionary attacks on the password. It uses a combination of asymmetric and symmetric cryptography. The asymmetric keys are ephemeral and are exchanged between the client and the server by encrypting it with the shared symmetric key — which is derived from the password. This allows securing the exchange against Man-in-the-middle attack. It is a symmetric PAKE so it requires that both party share a secret — namely the password. This means that the server has to store and process the password in cleartext which is strongly discouraged. Multiple cryptographic primitive can be used for the asymmetric part such as RSA, ElGamal or DH but the majority of EKE variants use DH [37]. Note that some of the variant got broken and EKE was patented until 2011 which might have severely impacted its adoption.

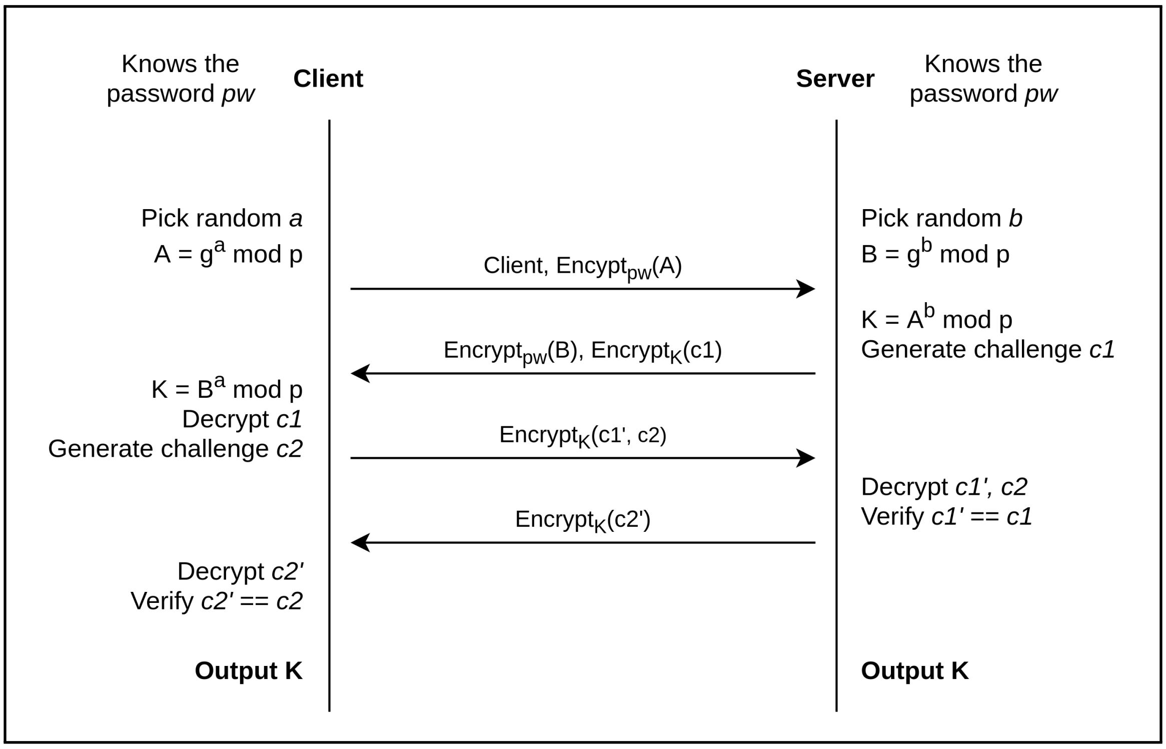


Figure 2.2: Login process with EKE (DH-EKE) protocol.

Construction. The figure 2.2 shows the EKE protocol — built with DH — during login process. The steps are the following :

1. Like a standard DH exchange, both client and server pick a random secret value a and b.
2. Client computes A, encrypt it using the password and send the result to the server in addition to it identifies (e.g., username).
3. Server decrypts ciphertext using the password to obtain A. He computes B and K. He encrypts B using the password and encrypt a randomly generated challenge $c1$ using K. He sends the resulting ciphertext to the client.
4. The client decrypts B using the password and compute K. He decrypts $c1$ with K and also generate a random challenge $c2$. He concatenate the two challenges, encrypt them using K and send the result to the server.
5. Server decrypt the ciphertext and check that both sent and received $c1$ match. If it is the case, the server is assured that the client possesses the same password. The server has authenticated the client. He finishes by encrypting $c2$ and sending the result.
6. Client decrypts the ciphertext and check that both sent and received $c2$ match. If it is the case, the client is assured that the server possesses the same password and therefore is authenticated. The client has authenticated the server.

Register. The protocol does not mention registration. It is assumed that both parties already share a common secret, the password. A secure channel is therefore necessary to share the password in the first place.

2.2.2 SRP

Introduction. SRP [36, 35] (for Secure Remote Password) was proposed in 1998 and is the most widely implemented PAKE protocol in the world [18]. It is largely used in iCloud Key Vault — which could make it one of the most widely used cryptographic protocols [18] considering the number of active Apple devices worldwide — and in 1Password's password manager [16]. It is well standardized and has numerous implementation in different programming languages. It is in fact a TLS cipher suite [31], implemented in OpenSSL. This success is partly due to the SRP's creators will to avoid patents — unlike most of the PAKE of its time — but also to avoid export restriction imposed by US law by not using any encryption schema [30]. Their goal was to provide a technology that improve the security of existing password protocols while keeping the ease-of-use of passwords. In other words, provide a drop-in replacement to the classical authentication methods where the implementation does not require a deep change in contrary to EKE where the shared secret — the password — need to be stored in cleartext on the server making it difficult to manage correctly. This makes SRP easy to implement for developers and transparent for the user. One of the main strengths of SRP is that the server does not store the cleartext password or the hashed password. Instead it stores a password verifier that is a discrete logarithm one-way function of the password.

Even though SRP is in interesting construction and does some things right, it is not ideal. It got broken and patched multiple time — current version is SRP v6a, which is not broken. It is vulnerable to pre-computation attack because the server sends the cleartext salt to the client at the start of the exchange. With the salt, an adversary could build a table of password hashes — a time-consuming process — before compromising the server making it able to retrieve passwords instantaneously upon server compromise. In addition, the construction is weirdly complex. The protocol mix addition and multiplication in calculation. Using both operations require a ring rather than a cyclic group. This requirement makes it impossible to easily transfer the integer-based algorithm to elliptic curves. This requirement, also makes it challenging to provide a formal analysis of SRP because "existing tools provide no simple way to reason about its use of the mathematical expression $v + g^b \mod q$" [30].

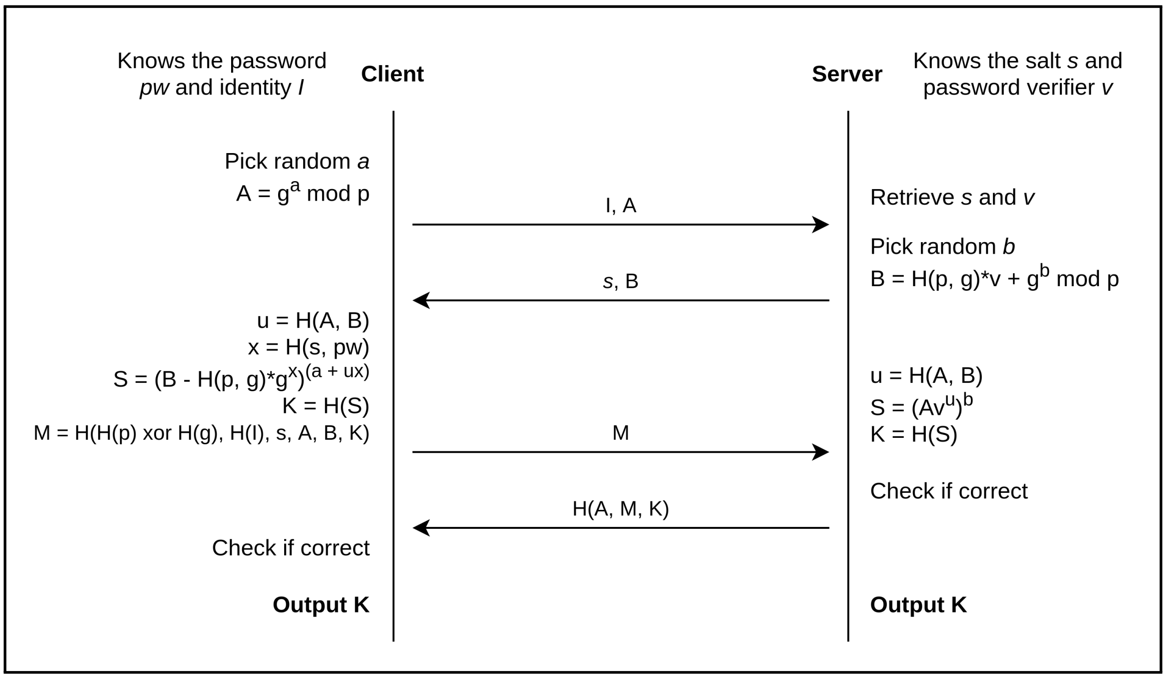


Figure 2.3: Login process with SRP-6a protocol.

Construction. The figure 2.3 shows the SRP-6a protocol during login process. The steps are the following :

1. Client picks a random a and computes A.
2. The client sends A and its identity I (username) to the server.
3. Server retrieves user salt s and password verifier v from its database using the user's identity.
4. v is computed at registration and is equal to $v = g^x$ where $x = H(s, pw)$
5. Server also picks a random b and computes B.
6. The server sends s and B to the client.
7. The client and server both compute u, S and K with their own values.
8. They finish with a mutual key-confirmation process where the client sends his proof of K first.
9. If the server finds that the user's proof is incorrect, he stops the exchange and does not send its own proof of K.

If the password is correct, the client and the server end up with having the same S — and so the same key. The derivation of S is less straightforward than other constructions so it is detailed below.

$$\begin{aligned}
S_{client} &\equiv (B - H(p,g)g^x)^{(a+ux)} \\
&\equiv ((H(p,g)g^x + g^b) - H(p,g)g^x)^{(a+ux)} \\
&\equiv (g^b)^{a+ux} \\
&\equiv g^{ab+bux} \\
&\equiv (g^a(g^x)^u)^b \\
&\equiv (Av^u)^b \\
&\equiv S_{server} \pmod p
\end{aligned}$$

Register. The registration process is not covered in SRP papers. Clients must come with a password, and server need to generate a random salt. One of them also need to compute the verifier $v = g^x$ where $x = H(salt, password)$. This means that either the server sends the salt to the client and the client compute the verifier and send it back to the server through a secure channel (more secure as the server never see the user's password but more complicated to implement). Or the client sends its password to the server with its registration request — through a secure channel — and the server generates a salt and computes the verifier (easier to implement on the server side, less transmission, but password is handled in cleartext on the server). Either way, the registration require to use a secure channel. Upon a successful registration, the server store the following triplet : `<username, verifier, salt>`

2.2.3 OPAQUE

Introduction. Jarecki et al. [23]. introduce the definition of Strong aPAKE (SaPAKE): an aPAKE secure against pre-computation attacks. They provide two modular constructions, called the OPAQUE protocol that allow building SaPAKE protocols. The first construction allows enhancing any aPAKE to a SaPAKE while the second allows enhancing any Authenticated Key-Exchange (AKE) protocol (that are secure against KCI attacks) to a SaPAKE. The security of these two construction is based on Oblivious PRF (OPRF) functions. The OPAQUE protocol allows to secure authentication from the simplest applications to the most sensitive ones.

OPRF. OPRF [14] is a two-party protocol that allows computing an output from two secret inputs. In other words, each party, namely the client and the server, input a secret value — a password for the client, a secret salt for the server — and the client can use the output as a key. What is interesting is that the client cannot learn the server's secret salt and the server cannot learn the user's password or the OPRF output. This feature allows OPRF to provide security against pre-computation attacks to OPAQUE and KHAPE. In addition to providing a Strong aPAKE protocol, OPRF provides other interesting security features that can be used with OPAQUE. Mainly, it supports the use of password hardening function, it allows an easy implementation of server threshold — instead of a single server providing its secret salt, multiple server would do it — and overall computing an OPRF provide a far more secure key than deriving directly from a password [23]. The figure 2.4 shows the OPRF process in the blue rectangle.

Construction. The figure 2.4 shows the OPAQUE protocol — built with OPRF and AKE — during login process. The steps are the following :

1. Generate a random value r to blind the hash of passwords so that the server cannot retrieve the password from the mapping.
2. Send result to the server.
3. Server add the salt to the password.
4. The client calculates the exponent of the inverse of r to de-blind the value. He cannot retrieve salt.
5. With the secret salt $salt_2$, client compute secret key sk.
6. Server send encrypted keys C to clients. C contains server's public key and client's private key encrypted with rw.

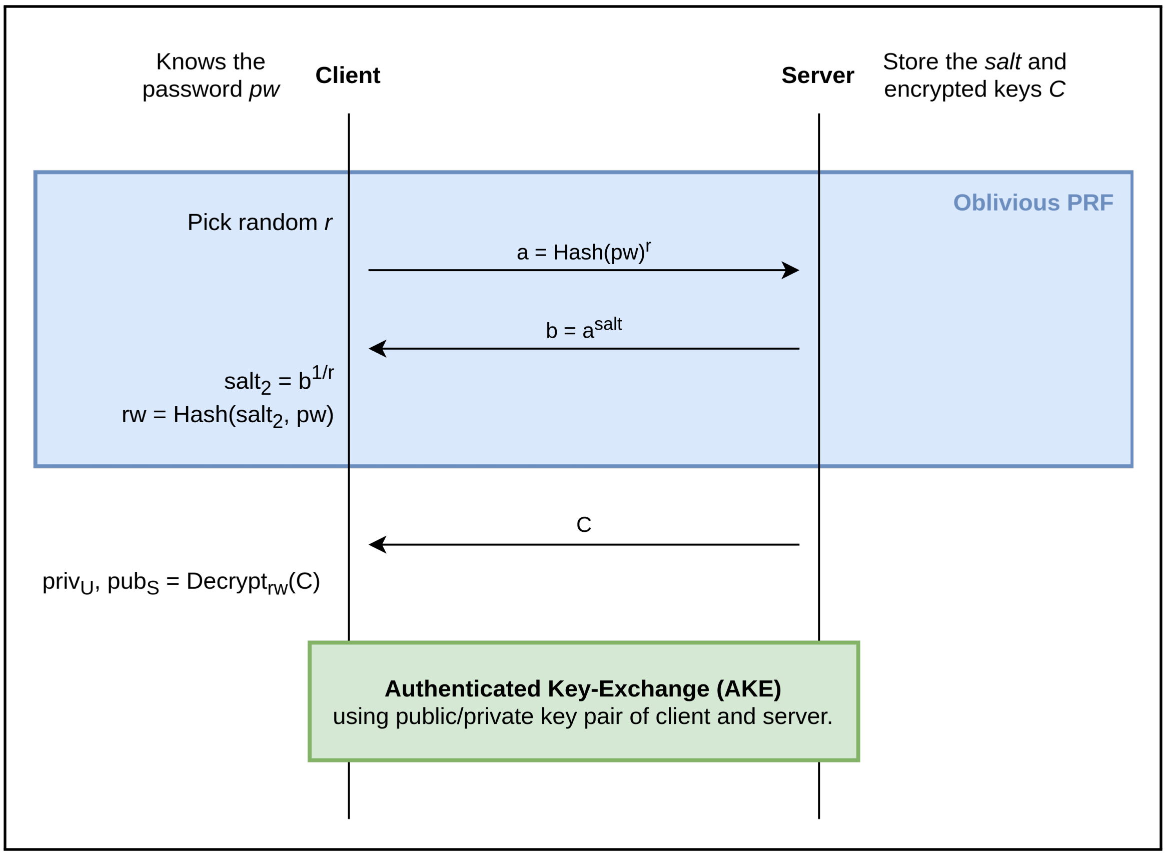

Knows the password *pw*
Client
Server
Store the *salt* and encrypted keys *C*
Oblivious PRF
Pick random *r*
a = Hash(pw)^r
b = a^{salt}
salt$_2$ = b$^{1/r}$
rw = Hash(salt$_2$, pw)
C
priv$_U$, pub$_S$ = Decrypt$_{rw}$(C)
Authenticated Key-Exchange (AKE)
using public/private key pair of client and server.

Figure 2.4: Login process with generic OPAQUE (OPRF-AKE) protocol.

7. If the password entered is correct, client uses rw to decrypt C and retrieve his private key $priv_U$.

8. With both keys, client and server run an authenticated key exchange for mutual authentication.

Register. The client registration is the only part of the protocol that requires a secure channel where both parties can authenticate each other. The protocol is proposed with a server-side registration where the client sends his password through the secure channel. The server generates a salt and computes OPRF function with the client's password and salt. Server also generates two private keys (one for the client and one for the server) and their corresponding public key. He encrypts client's private key and server's public key with OPRF output as a key and store the ciphertext. This method is not ideal as it requires that the user send its cleartext password to the server making it vulnerable to miss-handling or server-side vulnerabilities discussed in the introduction. OPAQUE's paper [23] mention that ideally, one wants to implement a client-side registration where the client choose a password and the server choose a secret salt and input them in the OPRF function. The client generates a public/private key pair, and the server do the same. Server sends his public key to the client. Client encrypts his private key and server's public key using OPRF output as a key. He then sends the ciphertext to the server with his public key. This way, the server never see the cleartext password, the OPRF output and the client's private key. This is a major improvement in terms of security. However, this also comes with a downside as the server is no longer able to check password rules. This operation needs to be done on the client side.

Login. For the login phase, the client enters its password in the OPRF and the server send the ciphertext to the client. If the password entered is correct, the client can decrypt the ciphertext with OPRF output to obtain his private key and the server's public key. He then uses these keys to run an authenticated key exchange with the server. On the other hand, if the password is wrong, the OPRF output is totally different and the ciphertext decryption makes the keys incorrect and the server will refuse it during the key exchange.

Differences with the internet standard draft. OPAQUE's paper [23] and OPAQUE's internet standard [10] — which is still in a draft state and is susceptible to be modified — have some differences. The standard draft has evolved in the last three years and is now at version seven (15th iteration). The draft standard is now much more detailed than before. One of the major differences in the core protocol design is the encryption of the client's private key. In the paper, it is specified that the client uses an authenticated encryption scheme to encrypt and decrypt its credentials — i.e., the client's key pair and the server's public key — with the OPRF output as encryption key. In the draft standard, it is rather different. Firstly, the envelope does not contain the client's keys anymore. It only contains a nonce and an authentication tag. The nonce is used — with a derivation of the OPRF output — to derive an authentication key, and export key and a seed. The authentication key is used to verify the envelope,'s authentication tag to ensure that the envelope content has not been modified. The export key is exposed to the client for application specific usage. The seed is used to derive the client's key pair. Secondly, the envelope is not encrypted at rest anymore. It is only encrypted during transmission between the client and the server with a one-time pad encryption. The rest of the time, the envelope is stored in cleartext on the server. During registration, the client derives a masking key from the OPRF output and sends it to the server with the envelope. The server stores the envelope and the masking key. Then, at login, the server generates a masking nonce and derive a one-time pad key using masking keys and masking nonce. He encrypts the envelope and his public key with the one-time pad key and sends the ciphertext with the masking nonce to the client. The client can compute the masking key with the OPRF output to compute the one-time pad key and decrypt the envelope.

2.2.4 KHAPE

Introduction. OPAQUE security relies entirely on the strength of the OPRF. If OPRF gets broken — for example by cryptanalysis, quantum attacks or security compromise — an adversary can compute an offline dictionary attack on the user's password. This is especially critical considering that there are currently no known quantum-safe OPRFs. KHAPE (for Key-Hiding Asymmetric PakE) [21] is a variant to the OPAQUE protocol. Instead of using OPRF as a main tool to archive security, it becomes an optional part of the protocol and KHAPE use two other concepts to archive security: non-committing encryption and key-hiding AKE. KHAPE is not a Strong aPAKE like OPAQUE. But it can be made a SaPAKE following the aPAKE to SaPAKE compiler from [23] using OPRF. So OPRF is optional with KHAPE and just allow making it a SaPAKE. In addition, it also allows using OPRF features such as a server-side threshold implementation that does not require any change from the client (see all features in Section 2.2.3). If OPRF fails, KHAPE just loss these functionalities but the rest of the security remain in contrary to OPAQUE.

Security without OPRF. Like OPAQUE, in KHAPE each party has a private/public key pair that is used to compute the key exchange and the client store its private key and server's public key on the server, encrypted by his password. If no OPRF is used, an adversary could try to retrieve the ciphertext and the server's public key from a valid key exchange between an existing client and the server. He can then compute an offline dictionary attack using a list of password candidates to decrypt the ciphertext until he finds a match with the server's public key that he recorded sooner. To avoid these kinds of attacks, KHAPE use two main mechanisms: Non-committing encryption and key-hiding AKE.

Non-committing encryption. Non-committing encryption makes it impossible for an adversary to identify the key that was used to encrypt a ciphertext even with a list of key candidates containing the correct key. This implies that the decryption of the ciphertext under any possible key provide a valid plaintext (i.e., a valid asymmetric key pair that can be used in the key exchange).

Key-hiding AKE. The key-hiding feature makes it impossible for an adversary to use the key exchange output to identify the correct private/public key decrypted from the ciphertext using a list of password candidates. Even if provided with a full transcript of a key exchange between a client and a server and a list of key pair candidates — including the correct key pair — the adversary has no better chance than random to guess the right key pair that was used.

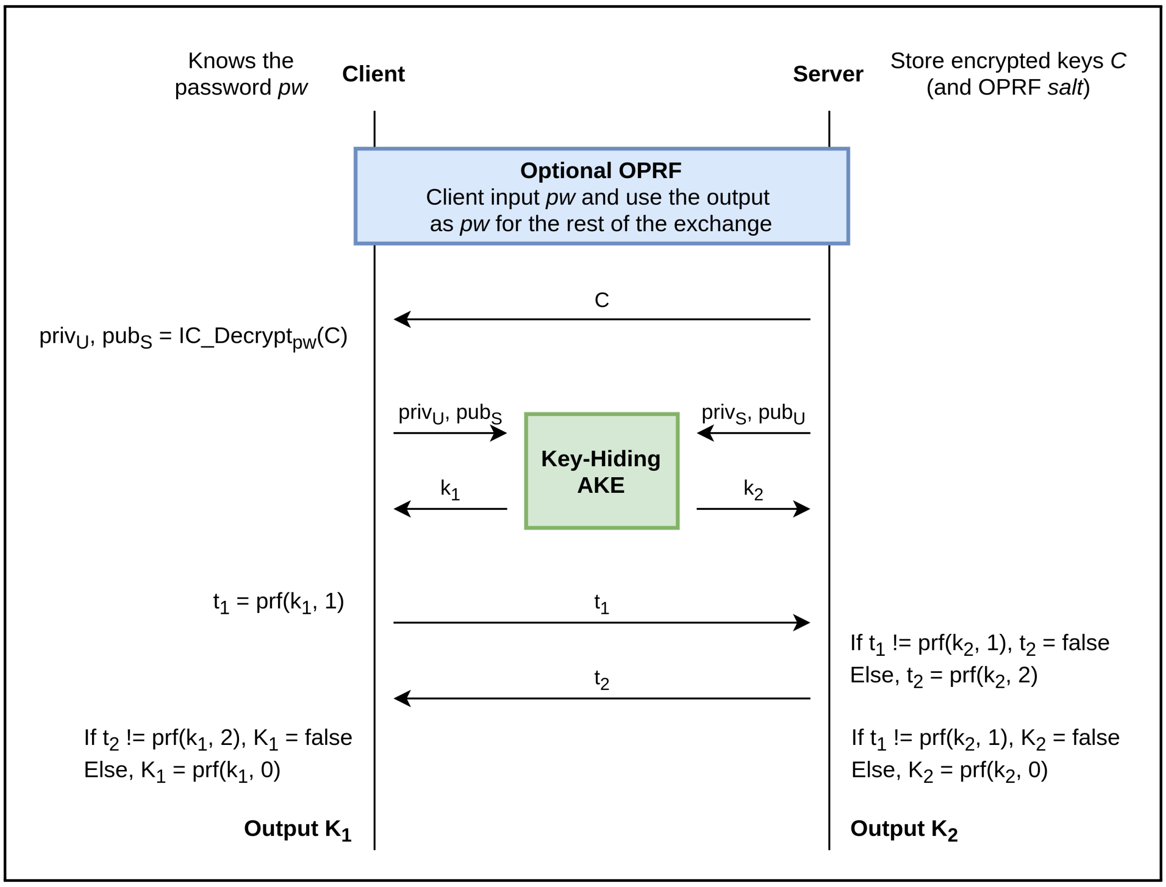

Knows the password *pw*
Client
Server
Store encrypted keys *C* (and OPRF *salt*)
Optional OPRF
Client input *pw* and use the output as *pw* for the rest of the exchange
C
$priv_U, pub_S$ = IC_Decrypt$_{pw}$(C)
$priv_U, pub_S$
$priv_S, pub_U$
Key-Hiding AKE
k_1
k_2
t_1 = prf(k_1, 1)
t_1
t_2
If t_1 != prf(k_2, 1), t_2 = false
Else, t_2 = prf(k_2, 2)
If t_2 != prf(k_1, 2), K_1 = false
Else, K_1 = prf(k_1, 0)
If t_1 != prf(k_2, 1), K_2 = false
Else, K_2 = prf(k_2, 0)
Output K_1
Output K_2

Figure 2.5: Login process with generic KHAPE protocol.

Construction. The figure 2.5 shows the KHAPE protocol during login process. The steps are the following :

1. Optionally, an OPRF can be used to archive Strong aPAKE following the aPAKE to SaPAKE compiler using OPRF from [23]. The OPRF takes the client's password and server's salt as an input. Client uses the output in place of his password for the rest of the protocol.

2. The server sends the client's encrypted envelope containing the client's private key and server's public key.

3. The client decrypts the ciphertext using Ideal Cipher encryption schema. He uses his password or OPRF output as a key.

4. Both parties use the public/private keys to compute a Key-Hiding Authenticated Key-Exchange.

5. Mutual key-confirmation process initiated by the client.

Login. When the client wants to login, the server sends its encrypted credentials and the client use its password to decrypt the credentials (with or without OPRF depending on the implementation). Then he can use his credentials to compute a Key-Hiding AKE with the server. Both party finish with a mutual key confirmation initiated by the client.

Register. KHAPE has the same problem than OPAQUE concerning the registration. The protocol proposes a server-side register which is less than ideal because the server can see the client's password and client's private key in cleartext at registration. Instead, it is recommended to use a client-side registration process. This problem and the solution is addressed in Section 2.2.3 for the OPAQUE protocol but also apply for the KHAPE protocol.

2.3 Comparing main solutions

This section compares the main PAKEs on their security guarantees and performances. Details and comments on each criterion can be found on Section 2.3.1 and the comparison table is presented on Section 2.3.2.

2.3.1 Details

1. Server does not process passwords in cleartext. This is the main security property of asymmetric PAKE [17]. Server does not have to store passwords in cleartext which should make it more resilient in case of server compromise. Adversary has to compute an offline attack to retrieve passwords from the compromised server.

2. Avoid sending cleartext password to the server. Even though it seems similar to the first criterion, it is not. The first criterion is about password processing, but this criterion is about password transmission. Transmission and processing of passwords are vulnerable to different attacks vectors. The server does not receive passwords in cleartext which avoid any miss-handling vulnerabilities such as logging or caching cleartext passwords on the server.

Comment: For OPAQUE and KHAPE, it may be required during register depending on the implementation (See Section 2.2.3).

3. Secure against pre-computation attacks. This is the main security property of Strong aPAKE [23]. The server does not leak any data that could allow an attacker to perform a pre-computation attack (for example SRP send the salt in cleartext in the first message). This attack allows an attacker to compute a table *before* the server even get compromised. Once the attacker succeeds in compromising the server, he can use the precomputed table to retrieve the passwords *instantaneously*. With this protection, an attacker can only perform an offline dictionary attack *after* successfully compromising the server. This protection is provided by the OPRF (see Section 2.2.3).

4. Forward secrecy. In key-exchange protocol, Forward Secrecy (also called Full Forward Secrecy or Perfect Forward Secrecy) ensures that upon compromise of any long-term key used to negotiate sessions key, an attacker cannot compromise previous session keys. In detail key-exchange protocol use long-lived keys to authenticate the user and short-lived keys to encrypt sessions. With Forward Secrecy, an attacker that successfully compromised a long-lived key cannot retrieve any previous session data even if he recorded the previous encrypted transmissions.

Comment: For EKE, only DH-EKE provide forward secrecy. ElGamal-EKE and RSA-EKE does not.

5. Mutual authentication. Mutual authentication explicit that users must be authenticated to the server but also that the server must authenticate itself to the user to avoid that an adversary impersonates the server to maliciously communicate with the client.

6. PKI-free. The transmission between client and server does not require to be secured with Public Key Infrastructure (PKI). This is a big improvement over classical authentication method (password-over-TLS) considering the occurrence of PKI failures nowadays.

7. User-side password hardening. Users can use password hardening technique to increase the cost of an offline attack if the server gets compromised. This is done by using resource-heavy functions such as Scrypt [28] or Argon2 [9] instead of computing a simple and efficient hash. These functions allow to drastically slows down hashing process and so making offline attacks and online guessing attack much slower.

Comment: For EKE, it could be possible to compute a KDF function on the password before using it as a symmetrical key but this is closer to Augmented EKE [7] where the password is hashed client side and the server store the hash results. For SRP, the client-side operation $x = H(salt||pwd)$ can be modified to use a resource-heavy hashing function [16].

8. Built-in mechanism to store client's secrets on the server. Securely store client's sensitive data such as secrets or credentials in the server without the server being able to read it. With OPAQUE and KHAPE, the user credentials (private/public keys) are encrypted with the password or OPRF output and then stored in the server. Additional secrets specific to the application could be added to this encrypted envelope and stored on the server.

9. Server threshold implementation. Require the interaction with n servers to authenticate. This means that n server has to be compromised in order for an adversary to compute an offline dictionary attack on the password. This scenario can be useful in the case of a highly sensitive application. OPRF transparently provides this functionality where each server adds its independent secret salt to the blinded password hash before sending it back to the client.

10. Resistant upon Oblivious PRF compromise. This criterion is a bit arbitrary because only the two recent PAKE uses an OPRF but it is still an important criterion because it is the main difference of security guarantees between OPAQUE and KHAPE. If OPRF breaks for example by cryptanalysis, security compromise or even quantum attacks, the consequences could be disastrous depending on the way it is used. This is especially important because there is “currently no known efficient OPRFs considered to be quantum safe” [21]. OPAQUE use OPRF as a main tool to builds Strong aPAKE. If OPRF breaks, the client’s password is vulnerable to an offline dictionary attack. KHAPE has a weaker reliance on OPRF. It is optional and only used to archive Strong aPAKE. If OPRF breaks, KHAPE only fall back to a non-strong aPAKE (making it vulnerable to pre-computation attacks). This makes KHAPE more resistant to OPRF compromise than OPAQUE.

11. Standardization status. The standardization status is a good indicator to the maturity and the interest in a construction. Standards make it easier for a developer to implement the construction since it is generally far more precise than theoretical paper and that some design choices are made in the standard. This allows interoperability between multiple implementation of the same construction — as long as they are following the standard.

12. Security proof. Security proof is an important factor to be able to put trust in a construction, but not all security proof are valuable. OPAQUE and KHAPE both provide security proof in strong models.

Comment: EKE only provide informal security analysis [5] SRP provide no valuable security proof [15, 19]. It only proves that it can stand up to passive attacks, which is not enough for an authentication protocol. A proof against active attacks would be welcomed for such a widely used protocol.

13. Easily adaptable to elliptic curves. Elliptic curve cryptography allows to greatly reduce the size of asymmetric key. This is crucial in terms of performance because key size recommendations are always increasing to ensure a high level of security. Asymmetric keys are getting giant and difficult to manage — in particular for keys that require long-term protection (up to 50 years). For example, for such a long-term protection, the discrete logarithm group is recommended to be 15,360 bits according to ECRYPT-CSA [3]. Elliptic curves, only require 512 bits to achieve a similar security level.

Comment: In SRP, $\mathbb{Z}_p$ is used as a field, not a group because it mixes addition and multiplication operations. Therefore, SRP cannot be easily adapted to elliptic curves [15]. For DH-EKE, it is required that the content that will be encrypted — namely A and B — must be indistinguishable from random data. This requirement makes it impossible to implement it on elliptic curves [12].

14. Number of messages. The number of messages has a direct impact on the performance of the algorithm — in particular on unstable or limited network. An increase in message number means a larger network latency and additional load on the network. Generally, it is preferred to have slightly larger message but fewer messages than the opposite due to all the networks overhead on packages.

15. Number of exponentiations. Exponentiations are generally the most time-consuming operation to compute in asymmetric cryptography. It makes it interesting to count the number of exponentiations to have a rough idea of the performances of the algorithm.

Comment: KHAPE with OPRF compute seven exponentiations where KHAPE without OPRF compute four.

16. Computational cost compared to a KE. The presentation of [21] provide a performance comparison between existing algorithms and KHAPE. It shows that without an OPRF, KHAPE-HMQV has the same cost as an un-authenticated key exchange. And with using an OPRF (OPAQUE), the cost is similar to the computation of two un-authenticated key exchange. SRP is not mentioned in the presentation but due to its complex exponentiations, it would probably be more costly than a single un-authenticated key exchange. Note that the cost of computation is theoretical and that the performance impact of using OPRF is not so blatant in practice. In fact, KHAPE without OPRF is around 25% faster during authentication and 45% faster during registration than with OPRF (see Section 6.3.2 for more details).

17. Communication size. Communication size can also have an impact on the perceived performance of the protocol — in particular on unstable or limited network. It depends considerably on the primitives used. Considering a group size suitable for near-term protection [3] — 256-bit elliptic curve and 3072-bit group — and 256 bits hashing function output size, we can simulate the communication size to compare them. EKE transmit two encrypted group elements and four encrypted challenges SRP transmit two group elements, one salt and two hash outputs. For OPAQUE and KHAPE, see section 6.6 for a detailed comparison of the communication size. For this table, KHAPE with OPRF is used. We can see that there is a considerable difference in communication size between EKE – SRP in one side and OPAQUE – KHAPE on the other side. This is because the latter use elliptic curve which greatly reduces the size of the underlying group representation without reducing the security level.

18. Server-side storage size. Server-side storage size does not have much impact on the perceived performance, but it is interesting to compare in terms of server administration. Considering that server can handle several thousand or millions of users, a small difference in storage per user can have a great impact on the overall storage required. To compare the algorithm, we use the same parameters as with the communication size. With EKE, the server only needs to store the cleartext password. With SRP, the server stores a salt and a verifier (group element). With OPAQUE, the server stores the OPRF's secret salt (32 bytes), the envelope (96 bytes) and AKE credentials (two group elements, 64 bytes). With KHAPE, the server stores the OPRF's secret salt (optionally, 32 bytes), the encrypted envelope (two group elements, 64 bytes) and AKE credentials (64 bytes). Similar to the conclusion made for the communication size, algorithm that does not use elliptic curves (SRP) require larger storage size.

2.3.2 Table

For result where there is a symbol "*", please refer to the criterion's precision marked with "*Comment*" in Section 2.3.1.

#	**Criteria**	**EKE**	**SRP**	**OPAQUE**	**KHAPE**
1	Server does not process passwords in cleartext	No	Yes	Yes	Yes
2	Avoid sending cleartext password to the server	No	Yes	Yes*	Yes*
3	Secure against pre-computation attacks	– (no hash)	No	Yes	Yes, with OPRF
4	Forward secrecy	Yes*	Yes	Yes	Yes

#	**Criteria**	**EKE**	**SRP**	**OPAQUE**	**KHAPE**
5	Mutual authentication	Yes	Yes	Yes	Yes
6	PKI-free	Yes, except during reg.	Yes, except during reg.	Yes, except during reg.	Yes, except during reg.
7	User-side password hardening	No*	Yes*	Yes	Yes, with OPRF
8	Built-in mechanism to store client's secrets on the server	No	No	Yes	Yes
9	Server threshold implementation	No	No	Yes, user-transparent	Yes, with OPRF
10	Resistant upon Oblivious PRF compromise	– (no OPRF)	– (no OPRF)	No, entire security is compromised	Fall back to non-strong aPAKE
11	Standardization status	RFC for EAP-EKE [29]	3 RFC [34, 33, 31], 1 ISO [2], 1 IEEE [1]	Internet standard draft [10]	CRYPTO 2021 Paper [21]
12	Security proof	No*	No valuable security proof	Yes, in the random oracle model	Yes, in the ideal cipher model
13	Easily adaptable to elliptic curves	No*	No*	Yes	Yes
14	Number of messages	4	4	3	4
15	Number of exponentiations	4	4	7	7 / 4*
16	Computational cost compared to a KE (see [21] presentation)	1x	>1x	2x	1x w/o OPRF, 2x w/ OPRF
17	Communication size	896 bytes	864 bytes	389 bytes	264 bytes
18	Server-side storage size	password size	416 bytes	192 bytes	160 bytes
19	Patented	Yes, expired in 2011	No	No	No
20	Year published	1992	1998	2018	2021
21	Got broken	Some versions got broken	Yes and patched [15]	No	No

3 | KHAPE

This chapter explain the details of the KHAPE protocol in terms of implementation and explain the design choices.

3.1 Choice of implementing KHAPE

The choice is based on the properties of the PAKE and the existence of implementations or standards for this PAKE. OPAQUE and KHAPE provide the highest level of security amongst aPAKE protocols. But while OPAQUE is becoming more mature with a draft standard and multiple high-quality implementations — including in rust **??**, KHAPE is still very recent. In fact, the KHAPE paper was published only six months ago at the time of the writing. To our knowledge, there is currently no public implementation of KHAPE and this is why we will be implementing it.

3.2 Generic algorithm

This section details a generic KHAPE protocol. Algorithms 5 and 6 show the pseudocode of a login process from both client side and server side. Algorithms 7 and 8 show the register process.

3.3 Design choices

This section explains the design choices made to implement KHAPE. Since the only resource for KHAPE is a rather theoretical paper, a large number of design choices have to be made. However, since OPAQUE and KHAPE share similarities, some implementation choices are inspired by the OPAQUE's standard draft whenever it is possible.

3.3.1 Client-side register

Both client-side and server-side registrations are possible but in order to use the aPAKE benefit to the fullest, it is preferable to handle the registration on the client. This allows to keep the main goal of aPAKEs: the server NEVER see the user's password. For more details, see Section 2.2.3.

3.3.2 Key Exchange

The PAKE protocol computes a key exchange to perform authentication. The key exchange has to be authenticated to avoid attacks such as Man-in-the-middle attacks that can be exploited on un-authenticated KE protocol like plain Diffie-Hellman. In AKE protocols, each party has two asymmetric key pairs. One long-term key pair that authenticate the exchange and one short-term (ephemeral) key pair that is used in the actual key exchange and only live for a single session. KHAPE requires that the underlying key exchange protocol is a "key-hiding" AKE (See Section 2.2.4 for more details about this construction). [21] shows that HMQV, 3DH and SKEME are key-hiding AKE.

OPAQUE's paper [23] and KHAPE's paper [21] shows concrete instantiation of their protocol using HMQV. These instantiations are easily adapted to other Diffie-Hellman based AKE like 3DH. Algorithms 1 and 2 show both key computation with HMQV and 3DH. We can see that the 3DH protocol requires to compute more exponentiation. In fact, between the two protocols, HMQV is more efficient but it is patented so 3DH is used for the KHAPE implementation.

Algorithm 1 HMQV protocol key computation for the client

```
Require: C := Client identity, S := Server identity
    $d_c \leftarrow$ Hash'(sid, C, S, 1, $X$)
    $e_c \leftarrow$ Hash'(sid, C, S, 2, $Y$)
    $o_c \leftarrow (Y \cdot B^{e_c})^{x+d_c \cdot a}$
    $k \leftarrow$ H(sid, C, S, $X$, $Y$, $o_c$)
```

Algorithm 2 3DH protocol key computation for the client

```
Require: C := Client identity, S := Server identity
    $o_c \leftarrow B^x || Y^a || Y^x$
    $k \leftarrow$ H(sid, C, S, $X$, $Y$, $o_c$)
```

3.3.3 Encryption scheme

KHAPE requires that the encryption scheme is non-committing (See Section 2.2.4). Non-committing encryption is achieved by combining an ideal cipher and a curve encoding.

Ideal cipher. As its name may suggest, the ideal cipher model is an idealized model used to prove the security of cryptographic systems. In practice, an ideal cipher is not implementable but it is possible to construct an encryption scheme that is indifferentiable from an ideal cipher. Multiple constructions have been investigated in the literature. The simplest construction is the one detailed in [22] which is an update from [13] that got broken. It is a 14-round Feistel construction that is indifferentiable from a random permutation. This construction is not focused on optimization and performance. Instead, every one of the 14 rounds are used in the security proof. Feistel construction allows building random permutation — the ideal cipher — from random functions in the random oracle model. The Feistel scheme function is $F_i = H(key|i|input)$. The hashing function must be large enough to receive half of the encryption envelope. Since this envelope store two group element, the hashing function must output a 256-bit value which is the size of a single group element.

Curve encoding. Before encrypting the group element with the ideal cipher, it is encoded as a bit string that is indistinguishable from random. This encoding “hides” the group element making it impossible for an adversary to identify the encryption key used for the ideal cipher — even if provided with a list of key candidates. This is because if a plaintext envelope is encrypted with key k and then decrypted with a different key k', the result is still a valid random group element that can be used in the key exchange. Since almost every ciphertext decryption provides a valid group element, an adversary cannot identify which encryption key is the correct one and is forced to compute the online key exchange with the server to validate its guess. This is done by implementing a quasi bijection from field elements to bit string. Elligator-squared [32] and Elligator2 [8] are implementations for such curve point encoding. Both these constructions are suitable for the implementation of KHAPE but since the elliptic curve used is Curve25519, it is more suitable for Elligator2.

Authentication. Generally, it is recommended to use authenticated encryption scheme because it is far more secure than un-authenticated encryption scheme. But for non-committing encryption, we must not authenticate it because it would break the main idea of non-committing. An adversary could test key candidates and compare the authentication tag, making it able to validate guesses.

3.3.4 Group $\mathbb{G}$

The key exchange is computed on a group $\mathbb{G}$ of prime order p with generator g. The group is generic which means that we are free to use an integer group or an elliptic curve group. For performances reasons (See Section 2.3.1), elliptic curves are used for the implementation. The curve must be compatible with the curve encoding algorithm selected (Elligator-squared, Elligator2, etc.). For current usage, it is recommended to use 256 bits elliptic curve [3]. For a more long-term usage (up to 50 years), it is recommended to use 512 bits elliptic curve. For the KHAPE implementation, curve25519 is used. It is a safe elliptic curve that provides sufficient security level and is well suited for implementing Elligator2 on it. Group elements are represented on 32 bytes.

3.3.5 OPRF

In KHAPE, the OPRF is optional — in contrary to OPAQUE — but allow making it a strong aPAKE resistant against pre-computation attacks (see Section 2.3.1). This is a major improvement in terms of security, and therefore an OPRF should be used. OPRF operations also need their own group and hashing function. There is also an ongoing standard draft protocol for OPRF [14] that presents multiple cipher suites for OPRF and different variants. The OPRF used for the KHAPE implementation is based on this standard draft using the ristretto255-SHA-512 cipher suite and the standard non-verifiable variant.

3.3.6 Slow Hash

It is possible and recommended to use a memory-hard hashing function (also called Slow Hash in this report) on the password to make it slow and expensive to compute. This makes it more costly for an adversary to compute hashes. For the implementation, Argon2id [9] is used because it is a recent memory-hard hashing function with a simple construction and it has better security analysis than other resource-heavy hashing functions like scrypt, bcrypt and PBKDF2 [15]. The Argon2id variant is used as it provides resistance against both GPU attacks and side-channel attacks — and is the recommended version. For OPAQUE, [10] propose to implement this function on the OPRF output rw and use the result to derive the encryption key. Section 3.3.7.1 shows how this hashing function is used in the key derivation process of KHAPE.

3.3.7 Key derivation

This section describes the process of deriving keys from existing secrets. The design is heavily inspired by OPAQUE's standard draft [10] since the context is similar. The primitive used to derive the encryption key, the export key, the output key and the key verification tags is HKDF [25]. It is well suited for expanding keys from existing secret and is already used in OPAQUE rust's implementation [26]. It follows the "Extract-then-Expand" paradigm where these two functions are used with the following API :

- Extract($salt$, ikm): Extract a fixed length pseudo-random key prk with high entropy from the input keying material ikm and the optional $salt$
- Expand(prk, $info$, L): Expand the length of an existing pseudo-random key prk with the optional string $info$ to produce an output keying material okm of L bytes.

HKDF is based on HMAC which in turn is based on a hashing function. The underlying hashing function should output at least 256 bits to fit the group element size. SHA-3 256 is used for its robust design.

3.3.7.1 Encryption key and export key

The OPRF output — or the password — is used to derive multiple keys with HKDF. The encryption key and the authentication key for computing authenticated encryption on the credential envelope. And the export key which is exposed at register and login and can be used by the application to encrypt application-specific data. In Chapter 5, it is shown how to use this key to encrypt user passwords for an online password manager. Algorithm 3 shows how these keys are derived.

Algorithm 3 KHAPE's encryption key and export key computation

```
Require: $rw$ := OPRF output
  $rw_{hardened} \leftarrow$ SlowHash($rw$)
  $rw_{randomized} \leftarrow$ Extract(salt="", ikm=concat($rw$, $rw_{hardened}$))
  $k_{encryption} \leftarrow$ Expand($rw_{randomized}$, "EncryptionKey", HashLength)
  $k_{auth} \leftarrow$ Expand($rw_{randomized}$, "AuthKey", HashLength)
  $k_{export} \leftarrow$ Expand($rw_{randomized}$, "ExportKey", HashLength)
  Output $k_{encryption}$, $k_{auth}$ and $k_{export}$
```

3.3.7.2 Output key and key verification

HKDF is also used to compute the output key K and both key verification tag t_1 and t_2. Instead of exposing the handshake secret k and computing each verification tag and the output key individually, these three values are computed at the same time and stored until needed. Algorithm 4 shows how these keys are derived.

Algorithm 4 KHAPE's output key and key verification computation

```
Require: $o$ := AKE output, $preamble$ := protocol's identities and messages
  $prk \leftarrow$ Extract(salt="", ikm=$o$)
  $k \leftarrow$ Expand($prk$, concat("HandshakeSecret", Hash($preamble$)), HashLength)
  $K \leftarrow$ Expand($prk$, concat("SessionKey", Hash($preamble$)), HashLength)
  $t_1 \leftarrow$ Expand($k$, "ClientMAC", HashLength)
  $t_2 \leftarrow$ Expand($k$, "ServerMAC", HashLength)
  Output $K$, $t_1$ and $t_2$
```

3.4 Limitations

In opposition with classical authentication method where the client only sends an authentication request and receives a response, KHAPE's registration and authentication processes require multiple messages (round-trip) between the client and the server. This adds new difficulties for both sides.

3.4.1 Client

For both registration and authentication, the client is required to send two messages. If the client is a web browser, he cannot only send a login form to the server. Behind-the-scene request has to be made to compute the two round-trip. This means that some JavaScript has to be implemented but this was known from the beginning that PAKE protocol requires a greater implication of the client code-wise.

3.4.2 Server

The server communicate with multiple clients at the same time so when a request comes, he needs to know from which client it is to be able to use the correct value. This is even more complicated because during both authentication and registration, the server generate some secret value in the first round trip and needs these values in the second round trip. The way to store this value is outside the scope of the

KHAPE library. It is up to the applications using the KHAPE's library to decide how to store these value in their system. However, we still provide possible solutions and the recommended one to store these values.

Authentication. During authentication, the server needs to store his ephemeral private key for the second round-trip. Applications using the KHAPE library can store this value in the following locations :

- Store this value on the server in the user datastore.
- Store this value on the server in a session datastore. This requires that client generate a session id and sends it with every request.
- Encrypt the value with a server secret and send it to the client who sends it back (comparable to JWT).

The KHAPE library is implemented to use the first solution since it is the easiest of the three — it doesn't require an additional session datastore. With this solution, a unique user cannot authenticate multiple time at the same moment. It is not a common requirement but if the application needs it, one should use the second solution. The third solution can be used but application developers should be extra careful that the ephemeral keys never leak.

Registration. During registration, the server generates his key pair and the secret salt. He sends the public key but the private key and the secret salt has to remain secret. Applications using the KHAPE library can store these values in the following locations :

- Preregister the client an incomplete file entry.
- Preregister the client. Store the available values on the server in the user datastore and specify that the registration is not completed. When the second registration round-trip arrive, the server store the rest of the value and marks the user registration as completed.
- Solutions two and three from the authentication

This time again, the first solution is recommended since it is the easier to implement. A user registration that is not completed should be overridable if another registration request comes with the same username.

Algorithm 5 KHAPE : Authentication on the client (generic algorithm)

```
Require: Knows username uid and password $pw$
  if OPRF then
      $r \leftarrow$ GenerateRandomNumber in $\mathbb{Z}_p$
      $h_1 \leftarrow \text{HashToGroup}(pw)^r$
      Sends authentication request to the server with uid and $h_1$
  else
      Sends authentication request to the server with uid
  end if
  $x \leftarrow$ GenerateRandomNumber in $\mathbb{Z}_p$
  $X \leftarrow g^x$
  if OPRF then
      Wait to receive $e$, $Y$ and $h_2$ from the server
      $salt_2 \leftarrow h_2^{\frac{1}{r}}$
      $rw \leftarrow \text{Hash}(salt_2, pw)$
      $(a, B) \leftarrow \text{Decrypt}(rw, e)$
  else
      Wait to receive $e$ and $Y$ from the server
      $(a, B) \leftarrow \text{Decrypt}(pw, e)$
  end if
  $o_c \leftarrow \text{KeyHidingAKE}(X, Y, B, x, a)$
  $k_1 \leftarrow \text{Hash}(\text{sid, C, S}, X, Y, o_c)$
  $t_1 \leftarrow \text{PRF}(k_1, 1)$
  Sends $t_1$ and $X$ to the server
  Wait to receive $t_2$ from the server
  if $t_2 \neq \text{PRF}(k_1, 2)$ then
      $K_1 \leftarrow$ False
  else
      $K_1 \leftarrow \text{PRF}(k_1, 0)$
  end if
  output $K_1$
```

Algorithm 6 KHAPE : Authentication on the server (generic algorithm)

```
Require: Store password file $file$ containing <$e$, $b$, $A$[, $salt$]>
  if OPRF then
      Wait to receive authentication request from the client with uid and $h_1$
  else
      Wait to receive authentication request from the client with uid
  end if
  $y \leftarrow$ GenerateRandomNumber in $\mathbb{Z}_p$
  $Y \leftarrow g^y$
  if OPRF then
      ($e$, $b$, $A$, $salt$) $\leftarrow$ file[uid, S]
      $h_2 \leftarrow h_1^{salt}$
      Sends $e$, $Y$ and $h_2$ to the client
  else
      ($e$, $b$, $A$) $\leftarrow$ file[uid, S]
      Sends $e$ and $Y$ to the client
  end if
  Wait to receive $t_1$ and $X$ from the client
  $o_s \leftarrow$ KeyHidingAKE($X$, $Y$, $A$, $y$, $b$)
  $k_2 \leftarrow$ Hash(sid, C, S, $X$, $Y$, $o_s$)
  if $t_1 \neq$ PRF($k_2$, 1) then
      $t_2 \leftarrow$ False
  else
      $t_2 \leftarrow$ PRF($k_2$, 2)
  end if
  Sends $t_2$ to the client
  if $t_1 \neq$ PRF($k_2$, 1) then
      $K_2 \leftarrow$ False
  else
      $K_2 \leftarrow$ PRF($k_2$, 0)
  end if
  output $K_2$
```

Algorithm 7 KHAPE : Registration on the client (generic algorithm)

```
Require: Choose username uid and password $pw$
  if OPRF then
      $r \leftarrow$ GenerateRandomNumber in $\mathbb{Z}_p$
      $h_1 \leftarrow$ HashToGroup$(pw)^r$
      Sends registration request to the server with uid and $h_1$
  else
      Sends registration request to the server with uid
  end if
  $a \leftarrow$ GenerateRandomNumber in $\mathbb{Z}_p$
  $A \leftarrow g^a$
  if OPRF then
      Wait to receive $B$ and $h_2$ from the server
      $salt_2 \leftarrow h_2^{\frac{1}{r}}$
      $rw \leftarrow$ Hash$(salt_2, pw)$
      $e \leftarrow$ Encrypt$(rw, (a, B))$
  else
      Wait to receive $B$ from the server
      $e \leftarrow$ Encrypt$(pw, (a, B))$
  end if
  Sends $e$ and $A$ to the server
```

Algorithm 8 KHAPE : Registration on the server (generic algorithm)

```
Require:
  if OPRF then
      Waits to receive registration request from a client with uid and $h_1$
  else
      Waits to receive registration request from a client with uid
  end if
  $b \leftarrow$ GenerateRandomNumber in $\mathbb{Z}_p$
  $B \leftarrow g^b$
  if OPRF then
      $salt \leftarrow$ GenerateRandomNumber in $\mathbb{Z}_p$
      $h_2 \leftarrow h_1^{salt}$
      Sends $B$ and $h_2$ to the client
  else
      Sends $B$ to the client
  end if
  Waits to receive $e$ and $A$ from the client
  if OPRF then
      Store file[uid, S] $\leftarrow$ ($e$, $b$, $A$, $salt$)
  else
      Store file[uid, S] $\leftarrow$ ($e$, $b$, $A$)
  end if
```

4 | Implementation

This chapter define specific implementation details for the KHAPE library [1].

4.1 Parameters

Both the client and the server has to define their parameters. For a client and a server to authenticate, they need to have the same parameters. Generally parameters are fixed by the application developer and are the same for all clients of the application. They can also be negotiated during registration but this require the server to store each client specific parameters as it can be different. This use case is not in the scope of the KHAPE library.

OPRF It is strongly encouraged to use the OPRF. It provides a higher security level (see Section 2.2.3) without impacting too much on the performances (see Section 6.3.2).

Slow Hash It is also encouraged to use a Slow Hash function. It allows to be more resistant against attackers (see Section 2.3.1) but it has a high impact on the performances (see Section 6.3.3).

Overall, it is strongly discouraged from using neither of them because this mean that the credentials envelope is encrypted by a weak key resulting only of an HKDF derivation of the low-entropy password.

4.2 Exchanges

This section shows which function is called to produce each messages during the exchange between the client and the server for both registration and authentication.

[1]KHAPE's library implementation is available here : https://github.com/jul0105/KHAPE

4.2.1 Registration

The registration is processed on three messages. The client needs to keep the `oprf_client_state` and the server keeps `pre_register_secrets`. At the end of a successful exchange, the client can use the `export key` for application specific usage and the server stores the user's file entry.

```
Client                                                                   Server
------------------------------------------------------------------------------
(register_request, oprf_client_state) = client.register_start(password)

  ------------------------ register_request ------------------------>

(register_response, pre_register_secrets)
    = server.register_start(register_request)

  <----------------------- register_response ------------------------

(register_finish, export_key)
    = client.register_finish(register_response, oprf_client_state)

  ------------------------ register_finish ------------------------->

file_entry
    = server.register_finish(register_finish, pre_register_secrets)
------------------------------------------------------------------------------
Client can use export_key                               Server store file_entry
```

4.2.2 Authentication

The authentication is processed on four messages. The client needs to keep the `oprf_client_state` and the `ke_output` and the server keeps `server_ephemeral_keys`. At the end of a successful exchange, the client can use the `export key` for application specific usage and both parties has the same output key.

```
Client                                                                   Server
------------------------------------------------------------------------------
(auth_request, oprf_client_state) = client.auth_start(password)

  ------------------------- auth_request ------------------------->
```

```
(auth_response, server_ephemeral_keys)
    = server.auth_start(auth_request, &file_entry)

  <------------------------ auth_response -------------------------

(auth_verify_request, ke_output, export_key)
    = client.auth_ke(auth_response, oprf_client_state)

  ---------------------- auth_verify_request --------------------->

(auth_verify_response, server_output_key)
    = server.auth_finish(auth_verify_request, server_ephemeral_keys,
                         &file_entry)

  <---------------------- auth_verify_response ----------------------

client_output_key = client.auth_finish(auth_verify_response, ke_output)
-----------------------------------------------------------------------
client_output_key and export_key                      server_output_key
```

4.3 Function definition

This section shows the client and the server API. Each function is defined step by step to understand the operation computed.

4.3.1 Client

- `register_start(Password) -> (RegisterRequest, ClientState)`
  1. Compute OPRF initialization (optional)
  2. Build `RegisterRequest` with uid and OPRF blinded result
  3. Build `ClientState` with OPRF state
- `register_finish(RegisterResponse, ClientState)`
  `-> (RegisterFinish, ExportKey)`
  1. Generate asymmetric key pair
  2. Compute OPRF output (optional)
  3. Compute slow hash (optional)

4. Derive encryption key and export key
5. Encrypt envelope containing private key and server's public key
6. Build `RegisterFinish` with envelope ciphertext and own public key

- `auth_start(Password) -> (AuthRequest, ClientState)`
  1. Compute OPRF initialization (optional)
  2. Build `RegisterRequest` with uid and OPRF blinded result
  3. Build `ClientState` with OPRF state
- `auth_ke(AuthResponse, ClientState) -> (AuthVerifyRequest, KeyExchangeOutput, ExportKey)`
  1. Generate ephemeral asymmetric key pair
  2. Compute OPRF output (optional)
  3. Compute slow hash (optional)
  4. Derive encryption key and export key
  5. Decrypt envelope containing private key and server's public key
  6. Compute key exchange output
  7. Build `AuthVerifyRequest` with uid, verify tag and ephemeral public key
- `auth_finish(AuthVerifyResponse, KeyExchangeOutput) -> Option<OutputKey>`
  1. Verify server's verification tag
  2. Return output key

4.3.2 Server

- `register_start(RegisterRequest) -> (RegisterResponse, PreRegisterSecrets)`
  1. Generate asymmetric key pair
  2. Generate OPRF secret salt (optional)
  3. Compute OPRF evaluation with secret salt (optional)
  4. Build `RegisterResponse` with public key and OPRF evaluation
  5. Build `PreRegisterSecret` with private key and OPRF secret salt
- `register_finish(RegisterFinish, PreRegisterSecrets) -> FileEntry`

   1. Build storable `FileEntry` structure with encrypted envelope, server's private key, client's public key and OPRF secret salt

- `auth_start(AuthRequest, FileEntry) -> (AuthResponse, EphemeralKeys)`
   1. Generate ephemeral asymmetric key pair
   2. Retrieve encrypted envelope and OPRF secret salt from file entry
   3. Compute OPRF evaluation with secret salt (optional)
   4. Build `AuthResponse` with encrypted envelope, ephemeral public key and OPRF evaluation
   5. Build `EphemeralKeys` with ephemeral key pair
- `auth_finish(AuthVerifyRequest, EphemeralKeys, FileEntry)`
  `-> (AuthVerifyResponse, Option<OutputKey>)`
   1. Retrieve server's private key and client's public key from file entry
   2. Compute key exchange output
   3. Verify client's verification tag
   4. Build `AuthVerifyResponse` with own verification tag
   5. Return output key

4.4 Library choices

This section shows the dependencies to implement the KHAPE library and the choice of using them.

- `curve25519-dalek v3.2.0` : Pure rust implementation of group operations on Ristretto and Curve25519. Audited in 2019 [20]. Slightly modified to obtain full Elligator2 features (see Section 4.5.1).
- `voprf v0.3` : Implementation of a verifiable OPRF based on the standard draft [14].
- `sha3 v0.9` : Pure rust implementation of the SHA-3 (Keccak) hashing function.
- `hkdf v0.11` : Pure rust implementation of the HMAC-based Extract-and-Expand Key Derivation Function (HKDF).
- `argon2 v0.3` : Pure rust implementation of the Argon2 password hashing function.

- `rand v0.8` : Random number generators.
- `serde v1` : Framework for serializing and deserializing Rust data structures.
- `serde-big-array v0.3` : Helper to serialize large array. Used for the serialization of the ciphertext.

4.5 Interesting functions

In this section, we detail function that has interesting subtlety and that was not already deeply detailed in the last chapter.

4.5.1 Elligator element encoding

For the implementation of the curve encoding, Elligator2 is used. [8] defines how to implement and how to use this algorithm.

Implementation The rust library `curve25519-dalek` provides a partial implementation of Elligator2 for Montgomery curves. Partial because only the encoding of a field element to a curve point is implemented. The inverse function — the decoding — is not implemented. Fortunately, a pull request [2] for this library propose an implementation of the decoding function to be able to retrieve field elements from curve points. This pull request is not merged in the library at the time of the writing but we will still use it since KHAPE require both Elligator2 encode and decode. So for the implementation of the KHAPE library, a modified version of the library `curve25519-dalek` is used. It applies the pull request to provide the Elligator2 decoding function. It also makes available the `FieldElement` structure that is used for the encoding function's input and the decoding function's output but was reserved for the internal library usage. Note that this library has been audited [20] but the code of the pull request is obviously out of the scope of the audit and therefore one should be careful when using the KHAPE's library since it used unverified code. Still, the pull request only implement the operations detailed in [8] for the decoding function.

Usage [8] also define the usage to encode long-term Diffie-Hellman keys. The idea is to generate a random private key and compute the associated public key. Then, try to decode the public key with the Elligator2 mapping. If the decoding is successful, the outputted field element is used as the public key material but if the decoding failed, the key generation is restarted from the first step. Not all curve points can be mapped

[2]https://github.com/dalek-cryptography/curve25519-dalek/pull/357

with Elligator2. In fact, only half the points can be mapped, which means that the key generation has to be processed twice in average. Listing 4.1 shows how the key generation is implemented for the KHAPE library. It follows the pattern detailed earlier, where the public key is tested to be decoded and as long as the public key does not fit, the key generation is started over. Since the key generation is absolutely mandatory, there is no other exit condition to the loop than finding a valid key pair.

```rust
pub(crate) fn generate_keys() -> (PrivateKey, PublicKey) {
    loop {
        let private_key = generate_private_key();
        let public_key = compute_public_key(private_key);
        let result = elligator_decode(&public_key);
        if let Some(field_element) = result {
            return (private_key, field_element);
        }
    }
}
```

Listing 4.1: Key generation function

4.5.2 Rejection method

The rejection method [15] is used for the generation of the private key. This simple method defines that if a randomly generated value does not fit as an input to generate some cryptographic material, reject the random value and regenerate a new one instead of trying to make the value fit (i.e., computing the modulus, clear the most significant bit, subtracting, etc.). It is safer to generate random numbers this way as it preserves the uniformly random distribution of the value. However, the performance is impacted since the algorithm is generally computed multiple time before finding a valid value.

Listing 4.2 shows the usage of the rejection method for the KHAPE implementation. During the private key generation, a random 32-byte value is generated and made into a scalar type. Then, this value is tested if it can be reduced. The reduce function modify the value to make it fit to a valid private key but this is not what we want to use. Instead, we compare that the reduced value is not different than the initial value and if it is the case, we know that it is a valid private key. If not, the process start over with a new random value generation. In average, the process is completed in 6.8 tries.

```rust
fn generate_private_key() -> PrivateKey {
    loop {
        let private_key_candidate =
            Scalar::from_bits(thread_rng().gen::<[u8; 32]>());
        if private_key_candidate == private_key_candidate.reduce() {
            return private_key_candidate;
        }
    }
}
```

Listing 4.2: Private key generation function

5 | Use case

This chapter shows a practical use case of using the newly implemented KHAPE library and demonstrate the security benefits.

5.1 Context

This section details use cases where using an aPAKE provide advantages over a classical authentication method.

5.1.1 Online password manager

Online password managers are among the most sensitive applications out there because the leakage of users' data cascades into numerous compromised accounts on other services such as email accounts, social media, online banking. Using an asymmetric PAKE for an online password manager makes a lot of sense because the client does not have to disclose its master password to the password manager host. In other words, the client does not have to trust the password manager host not to decrypt its personal data or leak the master password (or any other intentional or unintentional miss-handling). In fact, multiple well-known online password manager such as iCloud Key Vault or 1Password uses an aPAKE (SRP).

5.1.2 Other use cases

More generally, using an aPAKE makes a lot of sense on applications where the server-side stored user data should not be visible to the server. This is the case for applications where the server does not process the user's data. For example, online backup, secure vault, password managers, etc. This is achieved with encryption and so require an encryption key for the client. Depending on the client, it is not feasible to store an additional symmetric key because it has to be securely stored — e.g., with an HSM — which cause problems of portability and key recovery. For example, for an online encrypted backup of a laptop or a smartphone, if the user loses its device, he cannot retrieve his online backup because the encryption key is stored on its lost device. For portability, the encryption key is typically derived from the user's password — the same password that he uses to authenticate with the server (you could require that the user input two different passwords but this is generally avoided because of bad user experience). Using a classical authentication method, the server store the user's encrypted sensitive data and also process the password in cleartext which is used to compute the encryption key. This void all the security of encrypting the sensitive data in the first place because the server — or a malicious party who compromised the server — could store the cleartext password, compute the encryption key and decrypt the sensitive user's data. This is the reason why aPAKEs are very interesting in this case scenario. The server **never** sees the user's password, so he cannot use it to decrypt user's data.

5.2 Design

This section describes the design of the use case: a multi-user online password manager. The basic concept is that the server stores the encrypted user's data. Each client can only access his personal encrypted data from the server. KHAPE is used for authentication between the client and the server. OPRF and SlowHash parameters are enabled to provide the highest level of security.

5.2.1 Encrypted user data

Each user stores his encrypted user data on the server. This data include an ENCRYPTED PASSWORD REGISTRY and an ENCRYPTED MASTER KEY. Every encryption is computed by the authenticated encryption scheme XChaCha20-Poly1305. The ENCRYPTED PASSWORD REGISTRY is a structure that contains the user's password. Passwords are double encrypted. The external structure — the PASSWORD REGISTRY — is encrypted, and then each individual password is also encrypted. Each encryption is performed with a different key, all derived from the master key.

The ENCRYPTED MASTER KEY is simply a key — the MASTER KEY — that is encrypted with another key — the KHAPE's export key. One could derive the MASTER KEY from the KHAPE's export key but this means that if the user wants to change his authentication password, it is necessary to decrypt and re-encrypt every single password entry and the PASSWORD REGISTRY with the new export key. This is not conceivable for a scalable password manager. Instead, in case of change in the authentication password, only the MASTER KEY is re-encrypted with the new export key. This idea is inspired from Bitwarden's online password manager design [4].

5.2.2 General process

The figure 5.1 shows the entire process of reading a protected password from the authentication request to the password decryption.

1. The user input his password in the client and the KHAPE authentication start (see section 3.2 for details on the process).
2. KHAPE authentication output a session key and an export key to the client. The session key is verified with the server. If the inputted password is invalid, no session key is outputted. An export key would still be outputted since it is not verifiable.
3. Client request to download its encrypted data from the server using the session key as an authentication token. The server also stores the session key and can verify that the client has been successfully authenticated. An authentication token expires after 24 hours.
4. If the authentication token is valid, the client receives his ENCRYPTED MASTER KEY and his ENCRYPTED PASSWORD REGISTRY.
5. He decrypts the ENCRYPTED MASTER KEY with the KHAPE's export key to obtain his private key.
6. He computes two HKDF Expand with the MASTER KEY and constant labels as input to obtain the EXTERNAL ENCRYPTION KEY and the INTERNAL ENCRYPTION KEY.
7. He decrypts the ENCRYPTED PASSWORD REGISTRY with the EXTERNAL ENCRYPTION KEY to obtain the INDEXABLE PASSWORD REGISTRY.
8. Now that the client has decrypted the external layer of the PASSWORD REGISTRY, he remove the following values from memory: EXPORT KEY, MASTER KEY, ENCRYPTED MASTER KEY and ENCRYPTED PASSWORD REGISTRY. He still has to keep the SESSION KEY to communicate with the server, and the EXTERNAL ENCRYPTION KEY to re-encrypt the PASSWORD REGISTRY upon modification.
9. In the INDEXABLE PASSWORD REGISTRY each entry's label and username are in cleartext but the password is still unreadable.
10. When the user chooses to read a password entry, the client uses the INTERNAL ENCRYPTION KEY to compute an HKDF Expand with the entry's label and username as context. The result is a key that is unique for this password.
11. The client finish by decrypting the encrypted password with the computed key. He obtains the readable password.

12. After sending the cleartext password to the user, the client remove the cleartext password and the INDIVIDUAL PASSWORD KEY from memory. This means that at rest the client only keeps SESSION KEY, EXTERNAL ENCRYPTION KEYS, INTERNAL ENCRYPTION KEYS and INDEXABLE PASSWORD REGISTRY in memory.

When the client adds, update or remove a password, he needs to update his ENCRYPTED PASSWORD REGISTRY and upload it to the server. This is done by computing an INDIVIDUAL PASSWORD KEY from the INTERNAL ENCRYPTION KEY and the password entry's label and encrypting the password with this key. Then the password entry is added or updated in the PASSWORD REGISTRY which is then encrypted with the EXTERNAL ENCRYPTION KEY. The resulting ciphertext is sent to the server.

5.2.3 Server endpoints

The server has only four endpoints:

- Registration (KHAPE)
- Authentication (KHAPE)
- File download
- File upload

Registration and authentication are handled by KHAPE's protocol (see Section 3.2). Upon successful authentication, the client can use the outputted session key as an authentication token. He needs to send his session token with every request to prove to the server that he is authenticated. The session token expires after 24 hours. File download and file upload allow the user to retrieve and commit his protected data from the server. The interactions between the client and the server are defined in the figure 5.2 for the file download endpoint and in the figure 5.3 for the file upload endpoint.

5.2.4 Client actions

The end user interacts with the client to access its passwords. At the client start, the user can either register or login. After a successful registration, he still needs to login. Upon successful authentication, the client automatically download the user's data file and the user can select one of the following actions.

- Read a password

- Add a new password
- Modify a password
- Delete a password

For actions where the password registry is modified (adding, modifying or deleting a password), the password registry is re-encrypted by the client (with the EXTERNAL ENCRYPTION KEY) and uploaded to the server. The interactions between the user and the client are defined in the figure 5.4 for reading a password entry, in the figure 5.5 for adding a new password entry, in the figure 5.6 for adding a modifying an existing password entry and in the figure 5.7 for deleting a password entry.

5.2.5 Advantages of KHAPE

Using KHAPE for the authentication has two main advantages. Firstly, and most importantly, it allows to authenticate the client to the server with a password without the password being ever visible to the server. As specified in the section 5.1, it is especially important for an application where the sensitive data is encrypted with a key that is also derived from the password. Secondly, the utilization of the export key for the password registry decryption makes it more secure and more efficient. More secure because computing an OPRF is more secure than just deriving a key in local since it forces the client to interact with the server. Only the server knows the secret salt used to compute the OPRF output. This mean that an attacker would be forced to compute online password guesses, making it easier to mitigate by the server. More efficient because KHAPE already compute an OPRF and a SlowHash for the authentication and derive multiple keys from the output — including the export key. If the export key was not provided to the application, the client would be forced to compute another SlowHash function on the password to derive a master key. This would drastically impact the performances of the application.

5.2.6 Security considerations

It is important to consider that even though each password is double encrypted, it is still possible to an attacker to retrieve all passwords. Since both the INDEXABLE PASSWORD REGISTRY and the INTERNAL ENCRYPTION KEY are kept in memory at rest, an attacker could dump the client's memory and derive every INDIVIDUAL PASSWORD KEY to decrypt all passwords. Mitigation to this would be to add a secret value to the INDIVIDUAL PASSWORD KEY derivation. This secret value would be stored outside of the memory in a secure location such as an HSM or a YubiKey. This way, even if an attacker can dump the client's memory, he cannot derive any INDIVIDUAL PASSWORD KEY and so he cannot retrieve any cleartext password.

5.3 Implementation

The online password manager has been developed [1] in Rust. It is a functional proof of concept that demonstrates the usefulness of KHAPE for such applications. The implementation is based on an existing online password manager project that was developed earlier this year by Gil Balsiger and me. Currently, the client is a CLI (command line interface). It is planned to be adapted The server uses an SQLite database to store user information. Each user's encrypted data structure is stored on the server in a separated file. The network between the client and the server is simulated meaning that the client simply call server's function instead of sending an actual TCP or HTTP request. In the future, it is planned to adapt this prototype to be usable in practice. This will be done by implementing the network part. The client will also be adapted to compile the rust code in WebAssembly. The final goal is to build a client for web browsers.

[1]Implementation can be found here : https://github.com/jul0105/OnlinePasswordManager

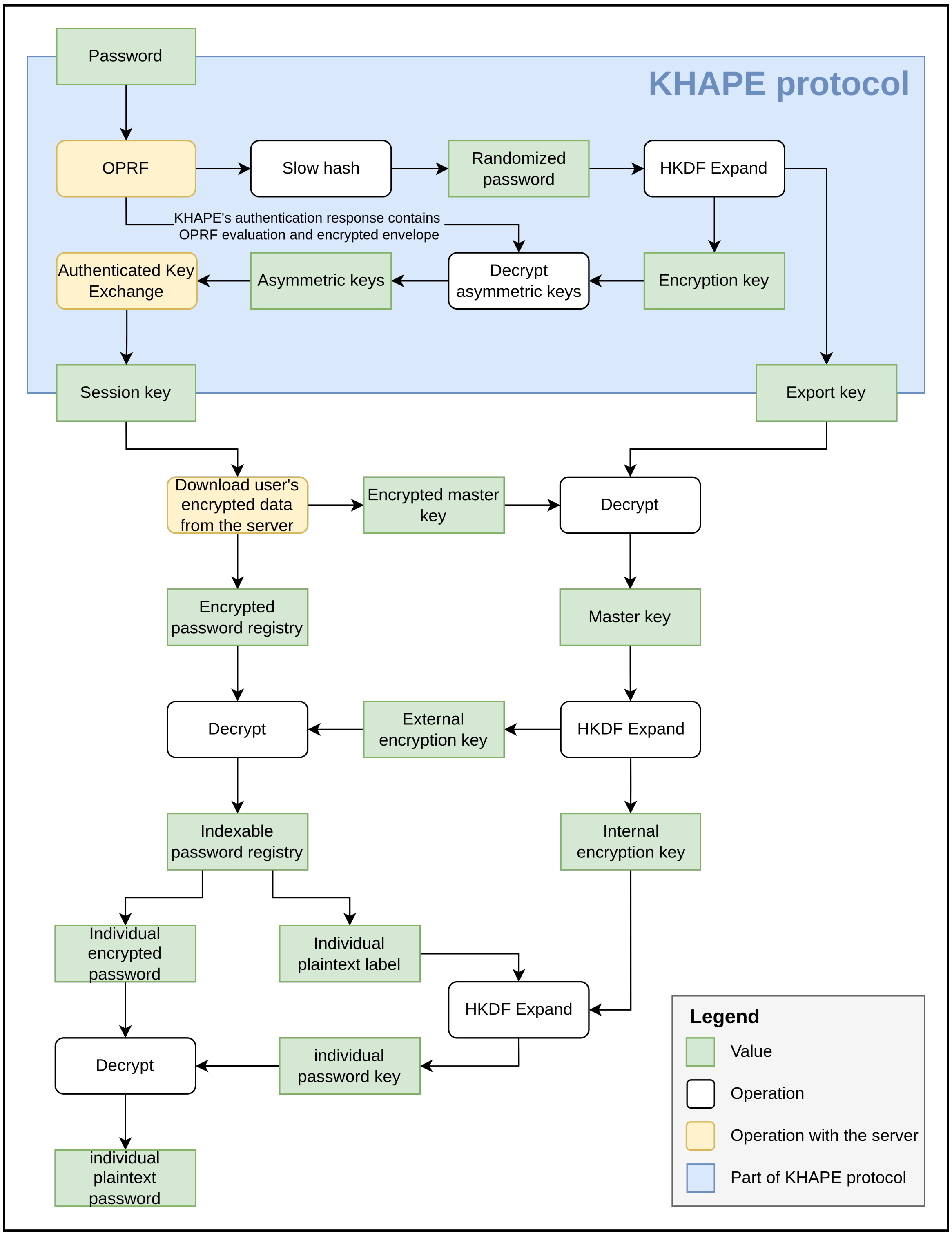


Figure 5.1: Online password manager key derivation process to read a password.

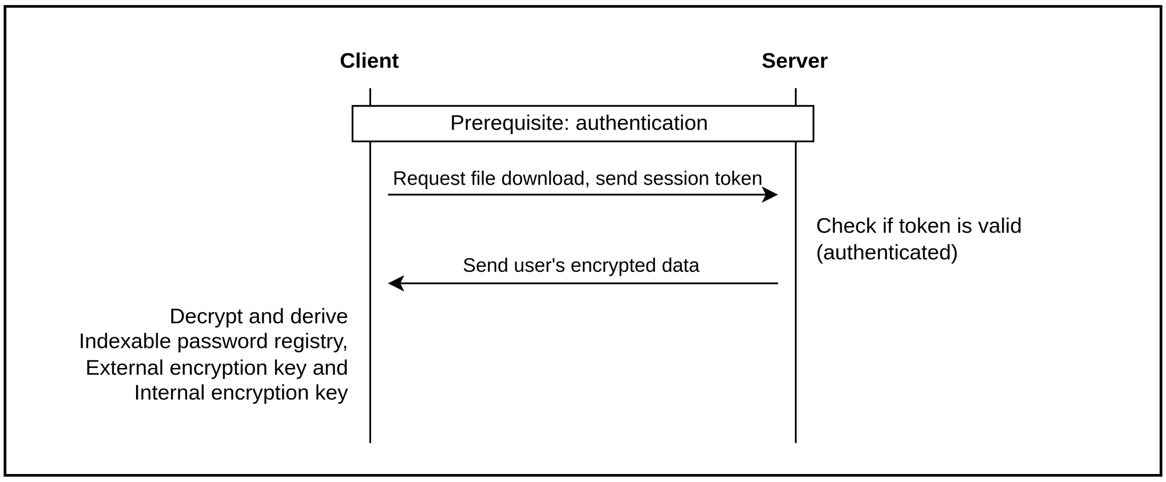

Client
Server
Prerequisite: authentication
Request file download, send session token
Check if token is valid (authenticated)
Send user's encrypted data
Decrypt and derive Indexable password registry, External encryption key and Internal encryption key

Figure 5.2: Interaction between the client and the server for the file download endpoint.

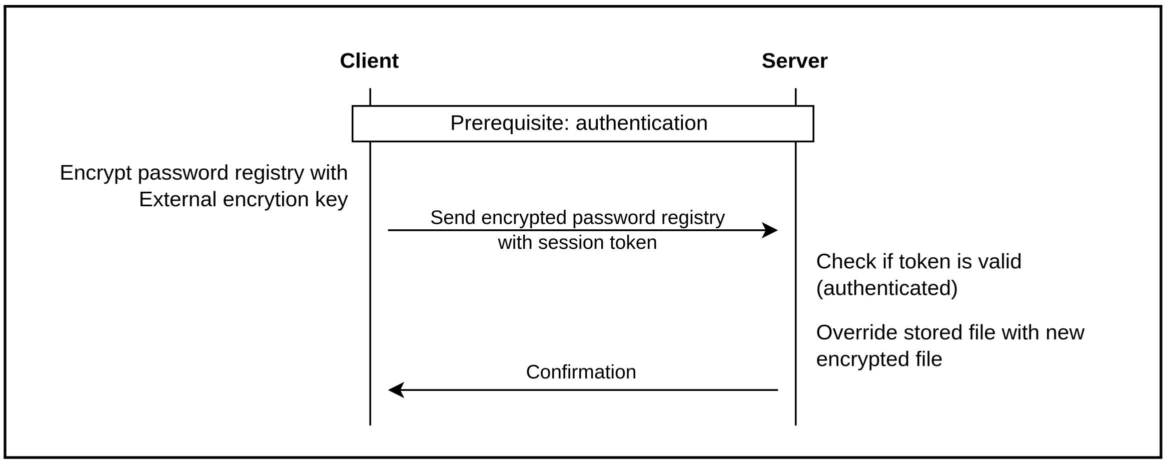

Client
Server
Prerequisite: authentication
Encrypt password registry with External encrytion key
Send encrypted password registry with session token
Check if token is valid (authenticated)
Override stored file with new encrypted file
Confirmation

Figure 5.3: Interaction between the client and the server for the file upload endpoint.

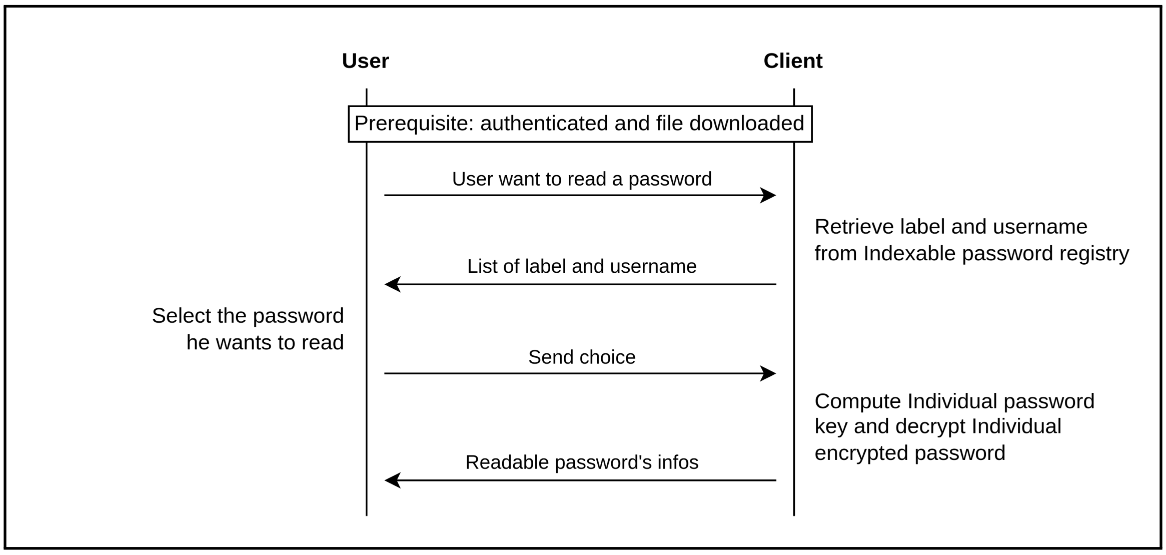

User
Client
Prerequisite: authenticated and file downloaded
User want to read a password
Retrieve label and username from Indexable password registry
List of label and username
Select the password he wants to read
Send choice
Compute Individual password key and decrypt Individual encrypted password
Readable password's infos

Figure 5.4: Interaction between the user and the client for reading a password entry.

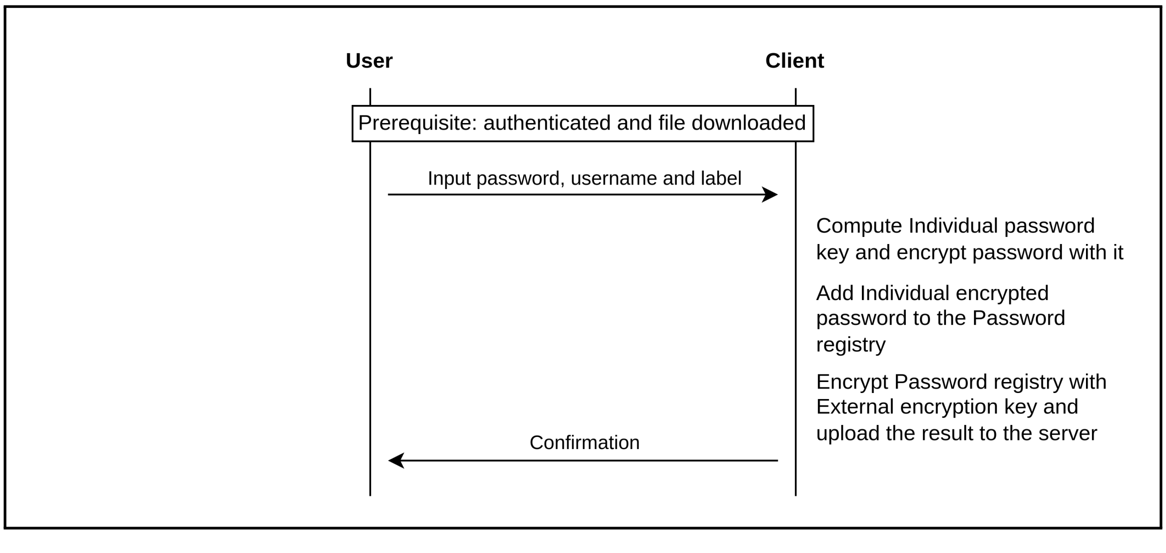

User
Client
Prerequisite: authenticated and file downloaded
Input password, username and label
Compute Individual password key and encrypt password with it
Add Individual encrypted password to the Password registry
Encrypt Password registry with External encryption key and upload the result to the server
Confirmation

Figure 5.5: Interaction between the user and the client for adding a new password entry.

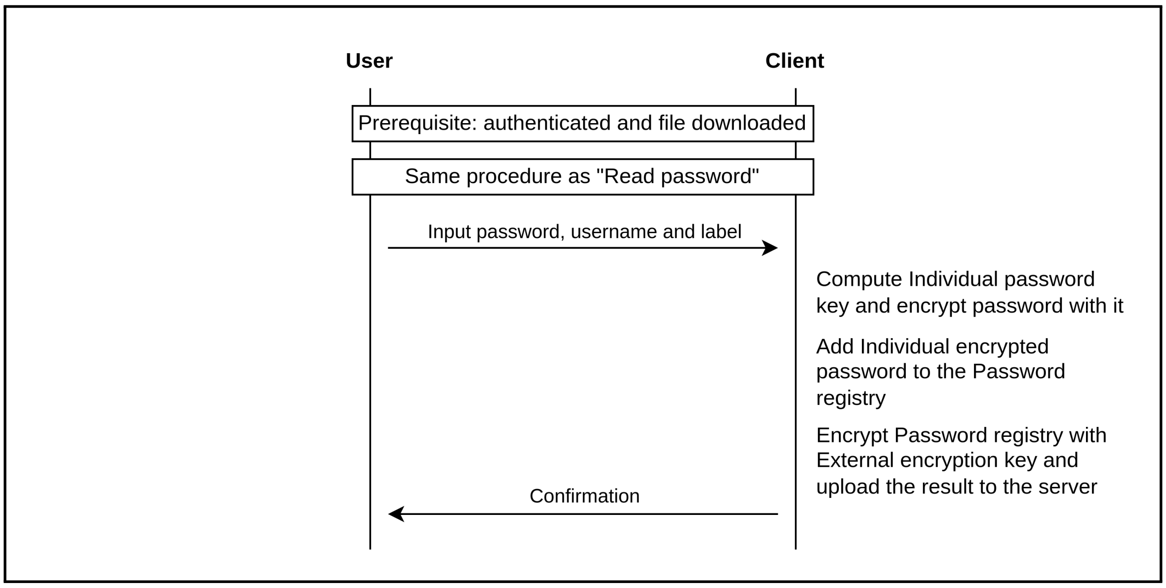

User
Client
Prerequisite: authenticated and file downloaded
Same procedure as "Read password"
Input password, username and label
Compute Individual password key and encrypt password with it
Add Individual encrypted password to the Password registry
Encrypt Password registry with External encryption key and upload the result to the server
Confirmation

Figure 5.6: Interaction between the user and the client for modifying an existing password entry.

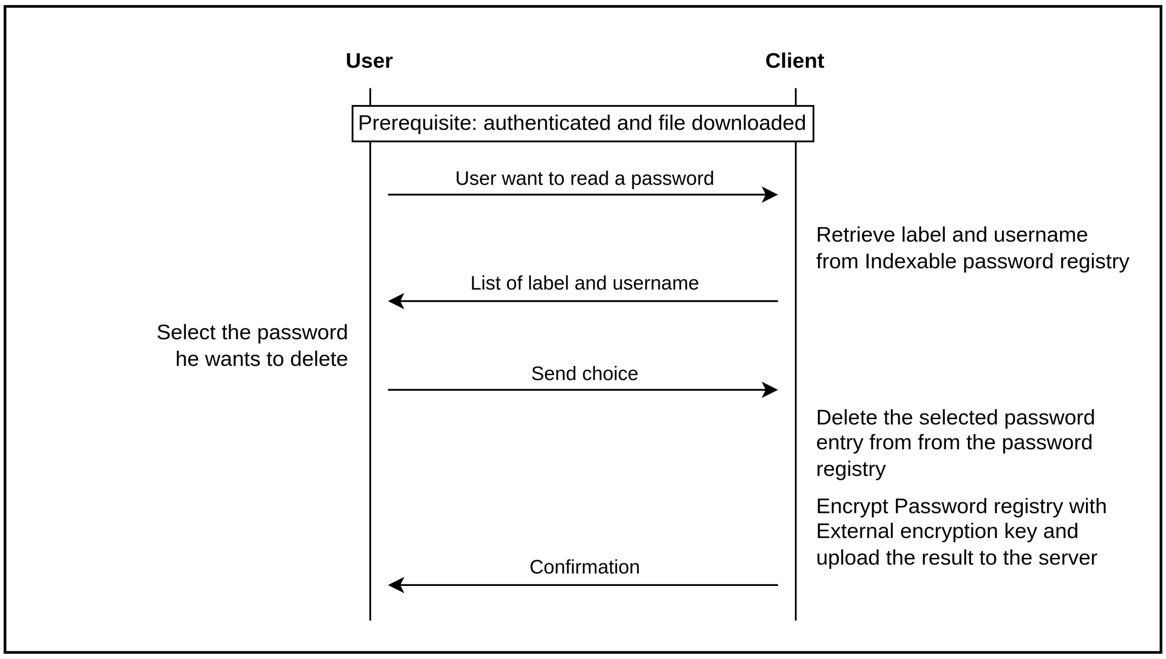

User
Client
Prerequisite: authenticated and file downloaded
User want to read a password
Retrieve label and username from Indexable password registry
List of label and username
Select the password he wants to delete
Send choice
Delete the selected password entry from from the password registry
Encrypt Password registry with External encryption key and upload the result to the server
Confirmation

Figure 5.7: Interaction between the user and the client for deleting a password entry.

6 Results

This chapter presents the performances obtained by the implemented KHAPE library using different parameters and analyzing the performance of some of the components. A performance comparison with the OPAQUE rust implementation [26] is made. The password manager use case is not in the scope of the test.

6.1 Testing environment

For computation benchmark, the rust library Criterion is used. It computes each benchmark 100 times. Each benchmark is sampled 100 times and each sample linearly increase the number of iterations on the tested function. The median is used for the charts. Benchmarks are computed on a HP Elitebook 850 G5 laptop with an Intel i7-8550U processor.[1]

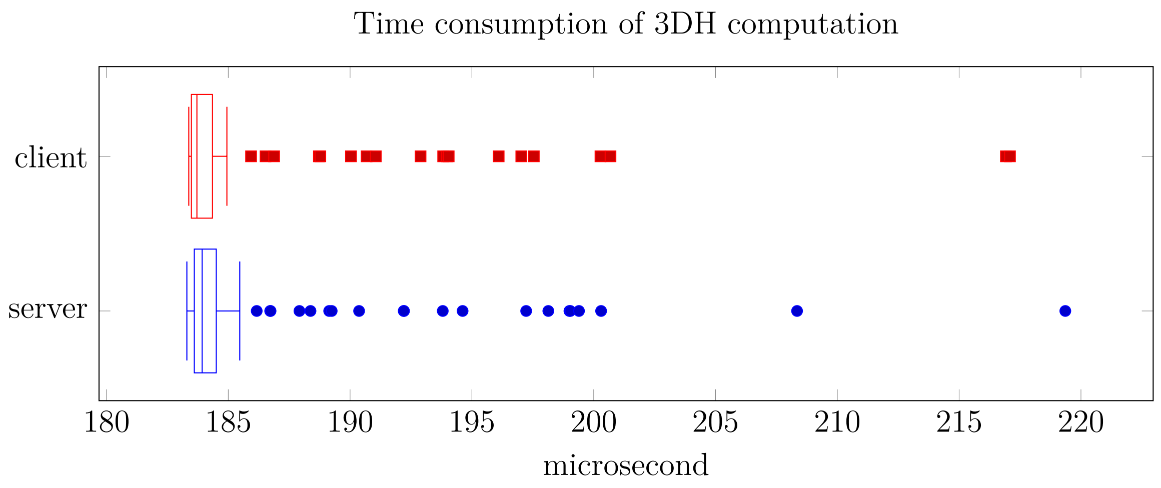
6.2 KHAPE components benchmark

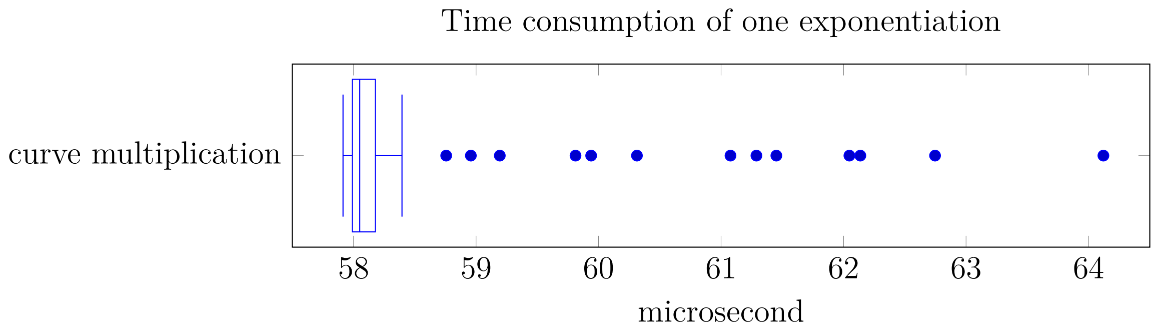
Before diving into the overall performances of KHAPE and its endpoints, it is important to understand what some of the components that take the most time to compute are.

6.2.1 3DH

The 3DH AKE is unsurprisingly the most time-consuming operation to compute. Considering that a curve multiplication takes around 58 µs to compute and that 3DH — as its name suggest — compute three of them. Adding the cost of deriving two keys using HKDF, the median time of 3DH computation is around 184 µs. Both client and server functions have around the same performances. This is not surprising because they compute the same operations with different party's key.

[1]The laptop is plugged to the charging cable during all the benchmarks. This makes a performance gain of around 40% compared to when it runs on batteries.

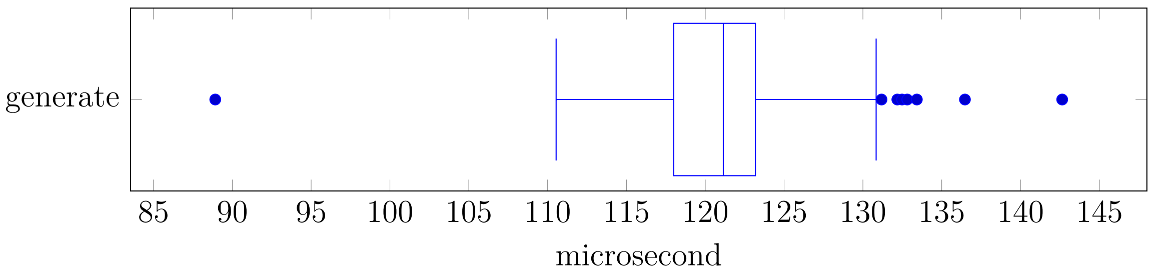




6.2.2 Key generation

Key generation also takes a lot of time with a median of 122 µs. Key generation consists of: 1) generating a 32 bytes random value, 2) generating a private key from the random bytes, 3) computing its associated public key — a curve multiplication operation — and then 4) converting the public key to a field element using the Elligator2 mapping. For step 2, not all 32 bytes random value can be used to generate a private key and for step 4, not all public key can be mapped to a field element. In this case, the rejection method is used which means that the key generation process is restarted from step 1. This is more secure than trying to modify the value to make it fit but on the other hand, it is more costly in terms of performance as the key generation process can be made multiple time. This is also why the time distribution of this function is more spread out than for other functions.

Time consumption of a key pair generation



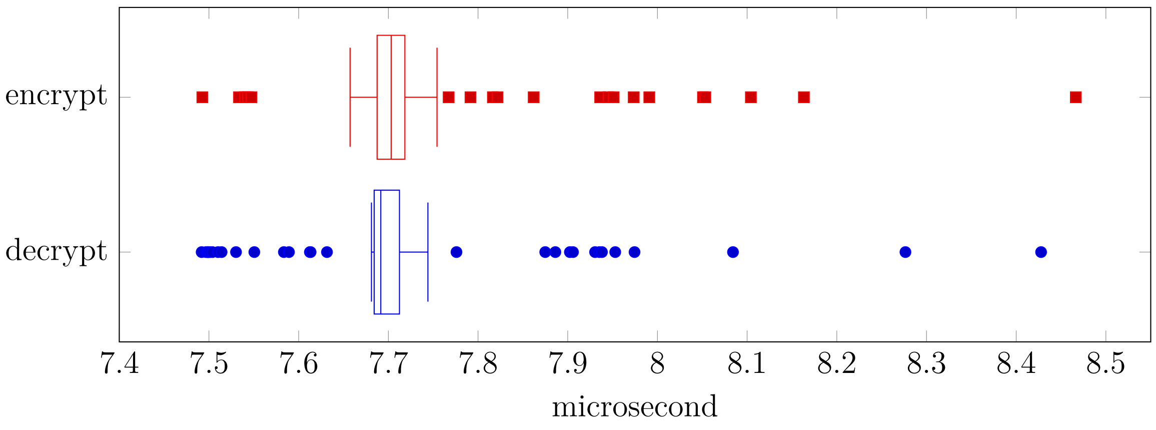
6.2.3 Encryption scheme

KHAPE require a Ideal Cipher encryption which has been implemented with a 14-round Feistel cipher.

Feistel cipher

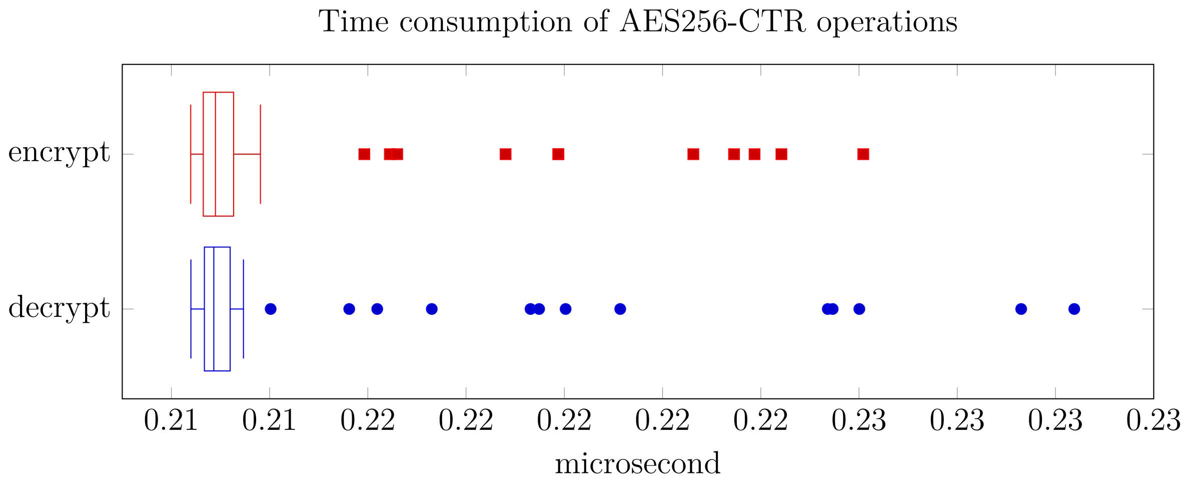
The Feistel cipher implemented is not optimal in terms of performance but it is surprising not that much time consuming. It is still around 36 times slower than AES256-CTR (see below) but it does not take much time compared to the group operation above.

Time consumption of Feistel cipher operations



AES256-CTR

Even though AES cannot be used for the encryption in KHAPE, it is still interesting to benchmark it with the same context that the Feistel cipher is used for. The encryption and the decryption take almost exactly the same time in median because with AES-CTR, the decryption is performed with the same function as the encryption.



6.3 KHAPE benchmark

In KHAPE, both registration and authentication process have multiple endpoints — respectively four and five. These endpoints are shared between the client and the server to constitute the protocol. For this benchmark, each endpoints are tested to measure the time they take to complete and compare them using different parameters. It is interesting to see Section 4.3 to understand what operations are computed on each endpoint and to compare it with the benchmark result. Note that only the time taken for each endpoint to complete is measured. In a real-world scenario, the resulting messages have to be transmitted between the client and the server. The network delay is not very interesting to benchmark as it depends too much on the network infrastructure between the client and the server. Instead, Section 6.6 shows a comparison of message size between multiple configuration of KHAPE and OPAQUE.

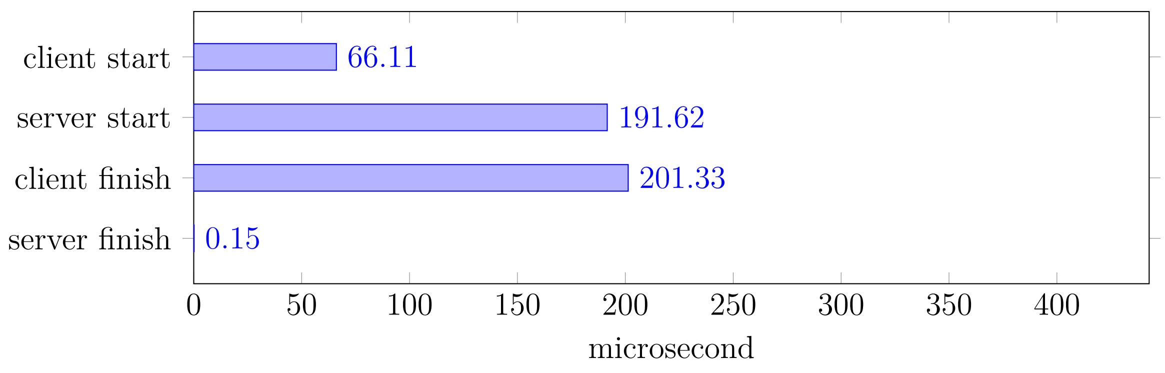
6.3.1 Standard configuration

KHAPE's default and most secure configuration is by using both OPRF and SlowHash. But since SlowHash functions are designed to be slow, it does not make sense to add it to the baseline benchmark because it does not show the real performances of the protocol. So for these benchmarks, the standard baseline configuration is KHAPE with OPRF and without SlowHash.

Registration

For the registration, we can see that both `server start` and `client finish` endpoints take the most time at around 200 µs. It is not surprising as it is where each party generate his own key pair, which take around 122 µs. The `server finish` endpoint take almost no time since it only builds a storable structure with the value received.

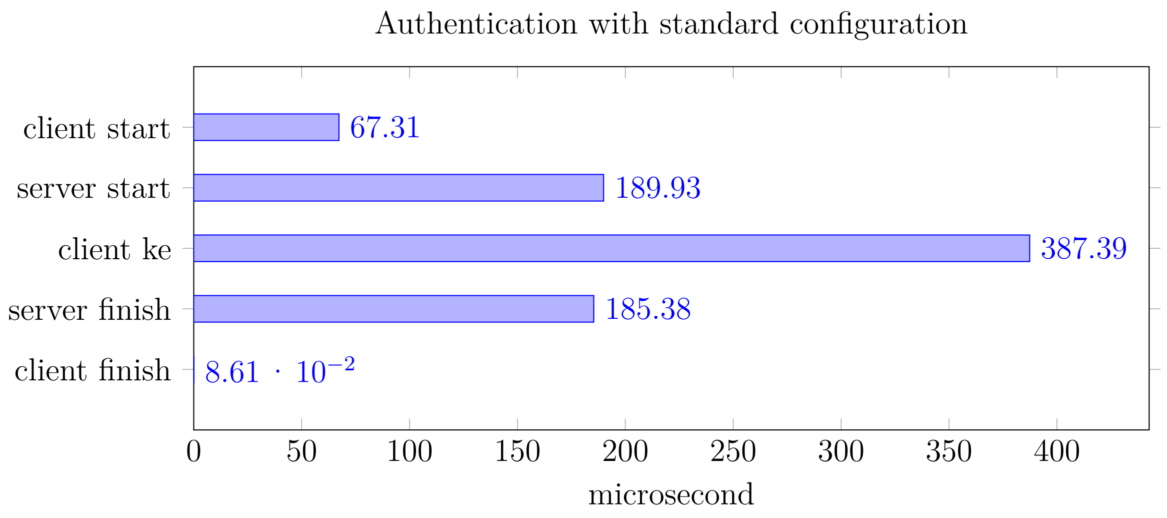
Registration with standard configuration



Authentication

We can see that `client ke` endpoint takes the most time. This is because a lot of things are computed in this endpoint including the generation of an ephemeral key pair (122 µs) and the computation of the 3DH (184 µs). We also notice that both `client start` and `server start` take about the same time as their homonym endpoints in registration. This is because these two functions are relatively similar between registration and authentication.[2]

[2]It is important to note that the endpoints for the registration and authentication are obviously different. For readability reason, it is not specified on the label of the charts but in the chart's title, which could make it look like authentication and registration share the same function.



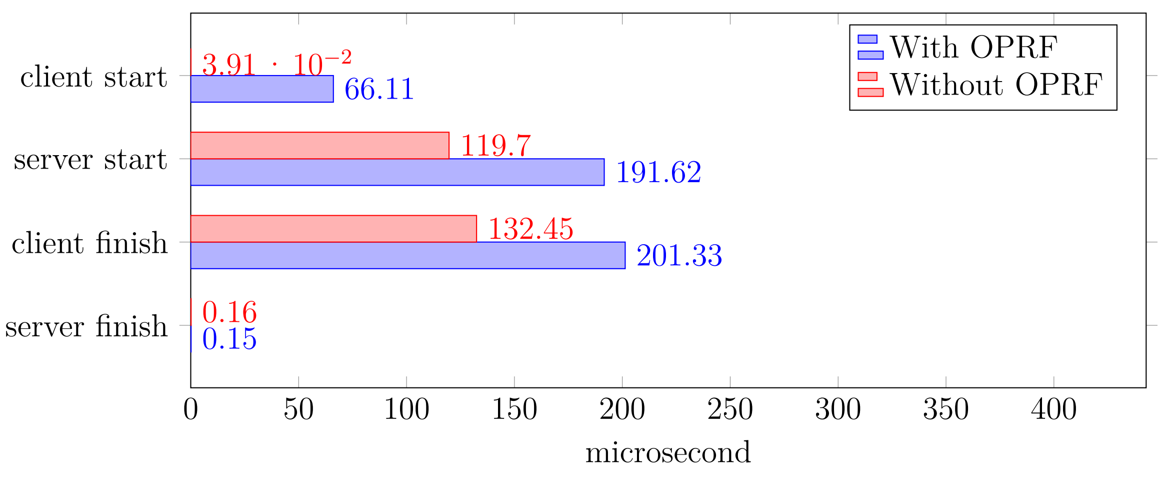
6.3.2 With OPRF vs. without OPRF

OPRF largely increase KHAPE security guarantees (see Section 3.3.5) but it remains optional. If used, OPRF is computed for both registration and authentication. It requires to compute two hashes and three exponentiations (curve multiplication) for each run. The exponentiation is computed on the first three endpoints of registration and authentication. In this section, we compare the performance of KHAPE with and without OPRF to see how much it impact the overall performances of the protocol.

Registration

Each OPRF concerned endpoints take between 66 and 72 µs more to compute when using OPRF. In total, the registration will take around 207 µs more than without an OPRF.

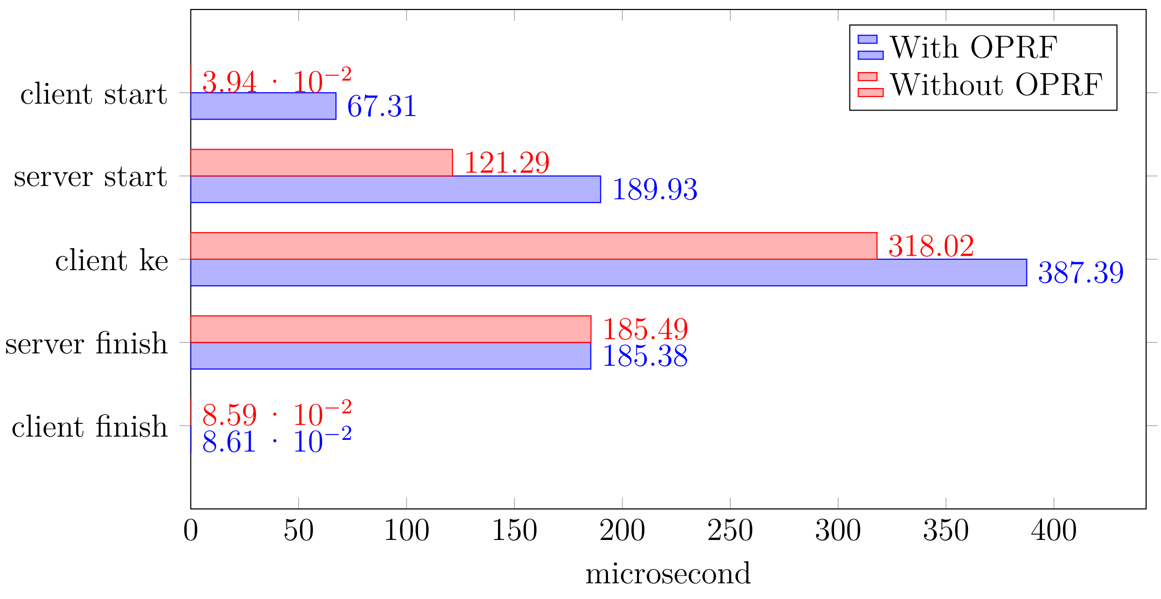
Registration with and without OPRF



Authentication

For the authentication, it is similar. The first three endpoints take between 67 and 69 µs more to compute and the total time addition for the authentication is around 205 µs more than without OPRF.

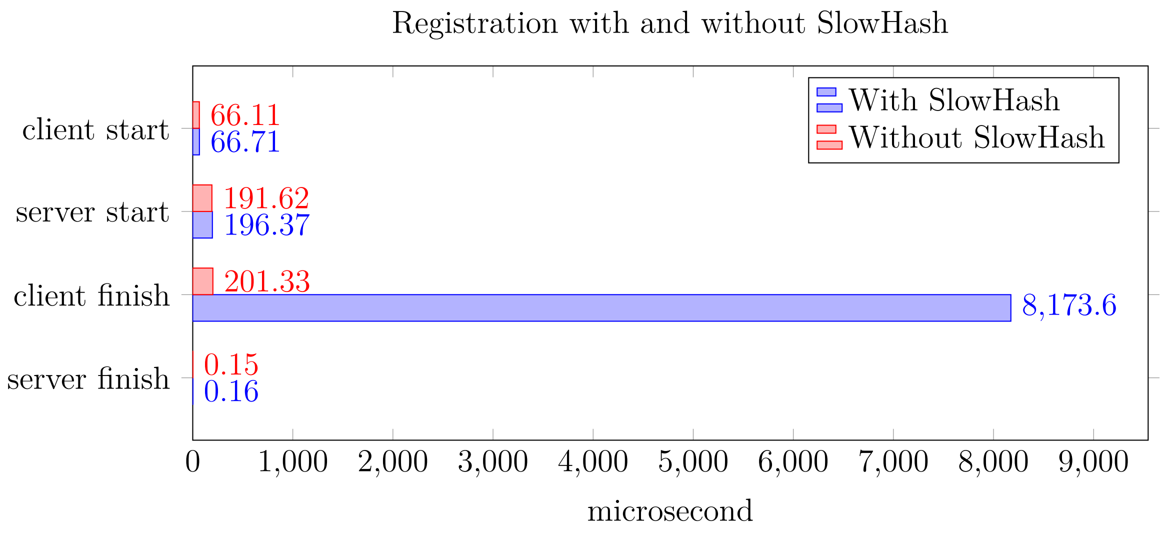
Authentication with and without OPRF



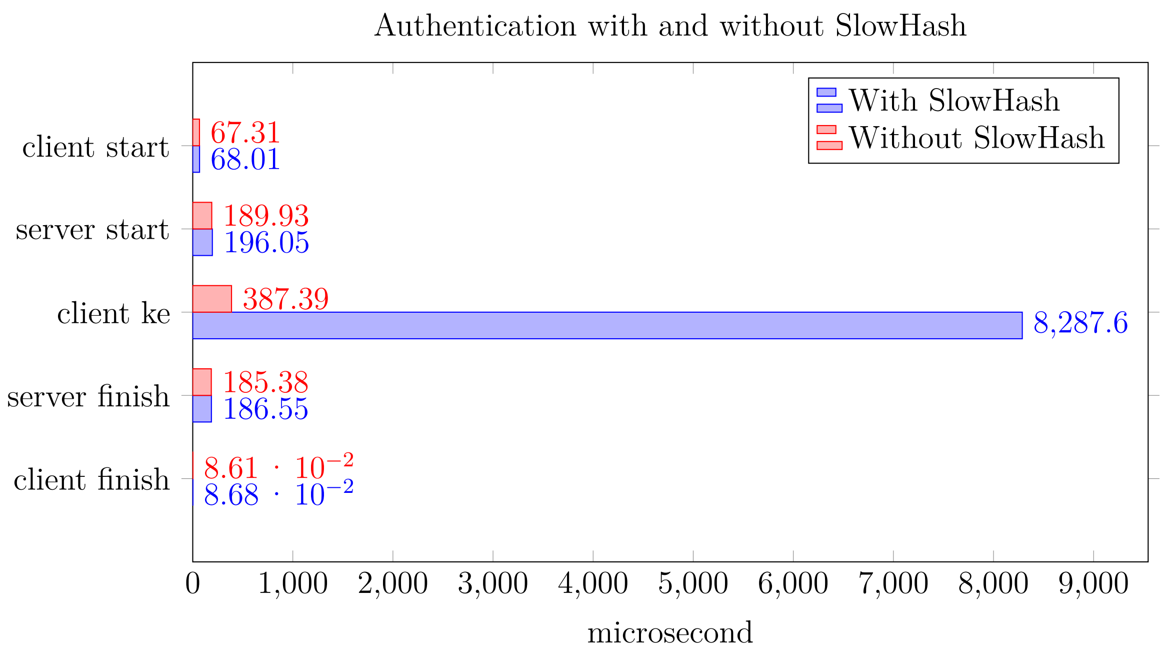
6.3.3 With SlowHash vs. without SlowHash

Using a memory-hard hashing function also increases the level of security of the protocol (see section 3.3.6). This benchmark is not very meaningful since these functions are designed to be slow and the performance only depends on the parameters used. But it is still interesting to see the performance degradation attached to the use of these functions and the scale of it compared to the rest of the protocol. It also makes it easier to understand how these functions slow down attackers that build hashing table. Since we replace a simple and efficient hashing function that would not even take 0.5 µs to compute by a slow and expensive hashing function that takes around $8,000$ µs to compute. That makes it 16,000 times slower to compute and so the attackers can compute 16,000 times less hash in the same time frame. This benchmark is using the rust library Argon2's default parameters. The default parameters of the KHAPE library are more resource intensive and time consuming since we believe that final user can wait a little bit more than 8 ms to register/authenticate to a secure system.

Registration



Authentication

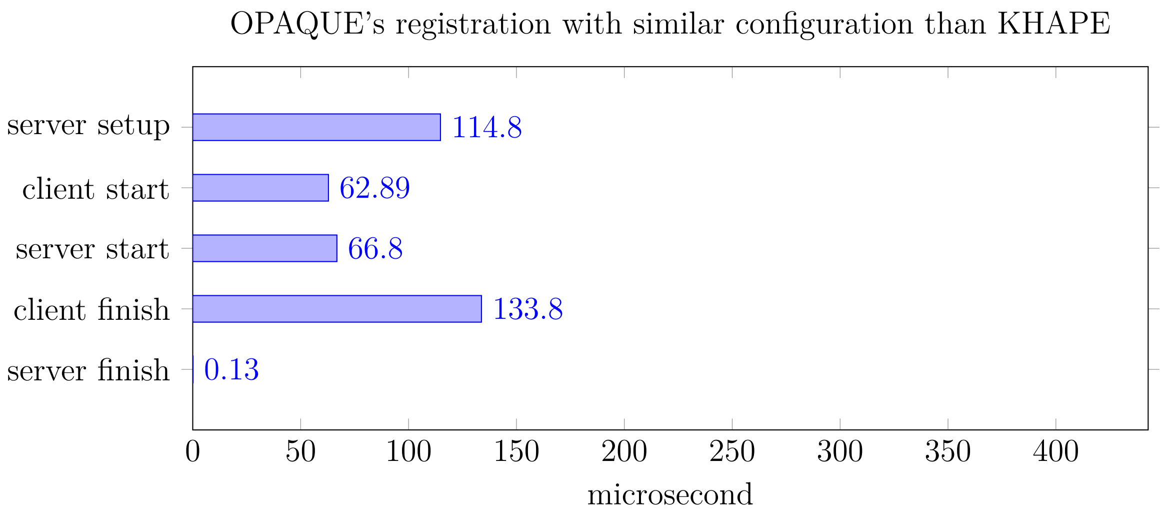


6.4 OPAQUE benchmark

KHAPE and OPAQUE designs share a large number of similarities. This makes OPAQUE a perfect candidate for performance comparison. But before that, we need to understand the difference to be able to compare them fairly. This benchmark is computed with a cipher suite that is close to the primitive used for KHAPE : ristretto255 curve for the OPRF group, Montgomery curve for the KE group, 3DH for the AKE and SHA3 for hashing. Note that the default cipher suite for OPAQUE use SHA2 instead of SHA3 and ristretto255 curve for both OPRF group and KE group but the performance obtained between the two cipher suites are relatively similar.

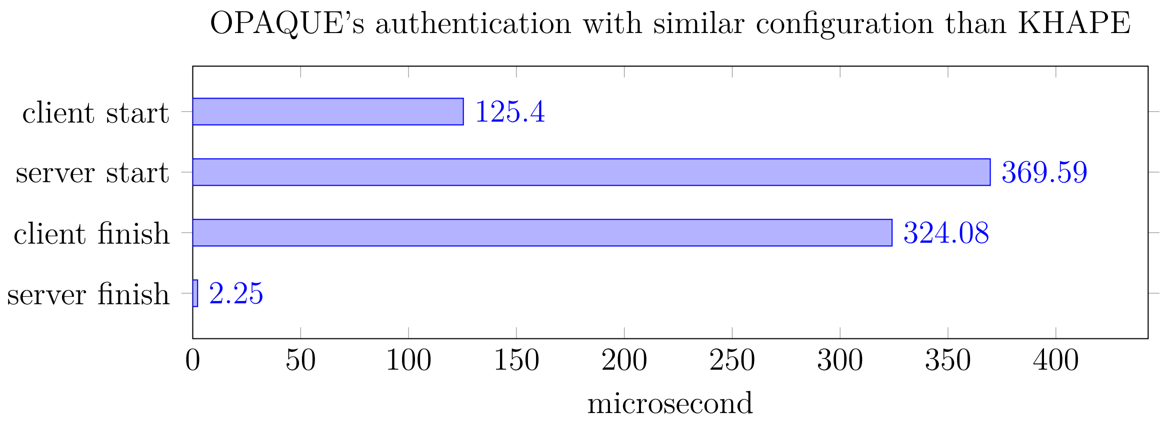
Registration

For the registration, OPAQUE has an additional endpoint `server setup` that generates the server's key pair. In KHAPE, this operation is done in the `server start` endpoint.



Authentication

For the authentication, OPAQUE has only four endpoints since the server initialize the key verification. The `server start` endpoint is the most time consuming with a median time of around 369 µs. It is responsible for generating an ephemeral key pair, compute the OPRF evaluation, computing the key exchange and deriving the output key and key verification tag. In KHAPE, all these operations are spread between `server start` and `server finish` endpoints.



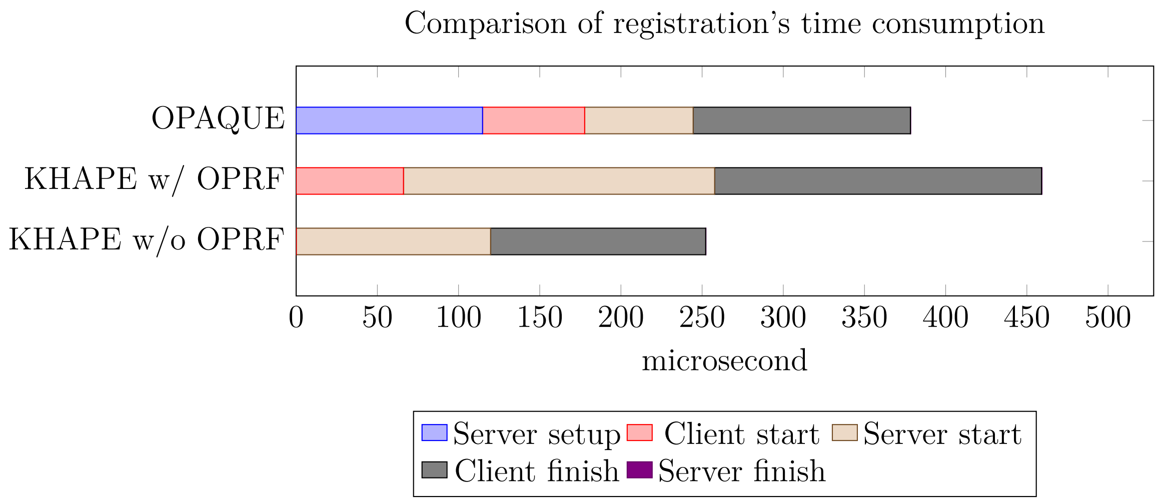
6.5 OPAQUE vs. KHAPE

In this section, we compare the performances of the KHAPE protocol — with and without OPRF — with the performances of the OPAQUE protocol.

Registration

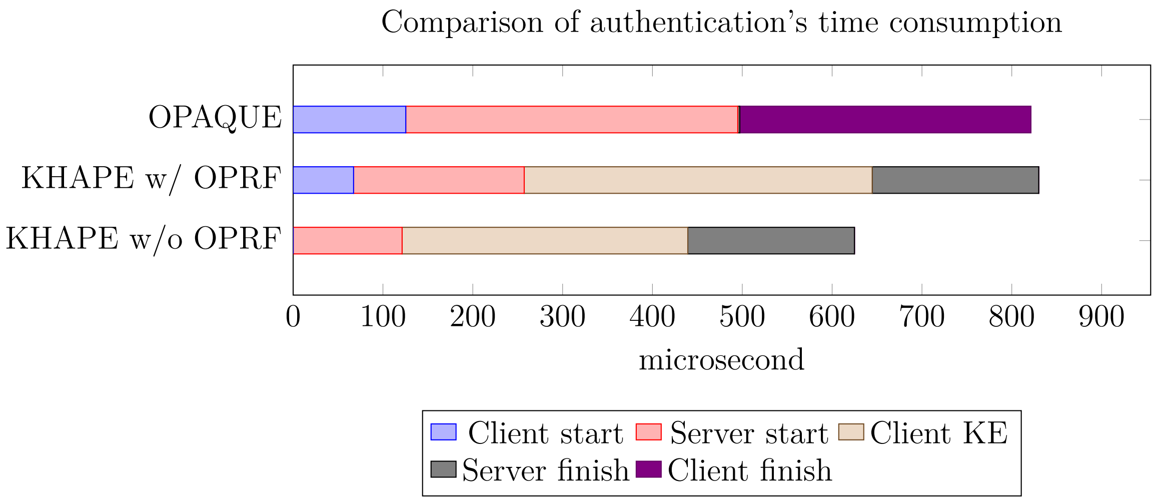
Firstly, we will compare OPAQUE and KHAPE with OPRF. For the registration, KHAPE with OPRF lose around 81 µs to OPAQUE in the median. The `client start` endpoints take about the same time. KHAPE's `server start` operations are split between `server setup` and `server start` for OPAQUE, which make it more flexible since it can be called before a request comes. In detail, it is the random key pair generation that is computed in the `server setup` endpoint. The two endpoints combined take about the same time as KHAPE's `server start` endpoint.

Majority of KHAPE time loss is on the `client finish` endpoint. This is probable due to the fact that KHAPE generate a key pair using the rejection method and then encrypt the private key and server's public key in an encrypted envelope. We saw earlier that encryption is not very time consuming (7.7 µs) but key generation is (122 µs). OPAQUE neither generate a key pair, nor use the rejection method nor encrypt the envelope. Instead of using the OPRF output to derive an encryption key to decrypt the encrypted private key, OPAQUE directly derive the private key (and the resulting public key) from the OPRF output. This solution allows OPAQUE to get rid of encryption altogether and to avoid computing a random key generation which makes it much faster. Also note that this method is only for the client side. The server still needs to generate a random key pair which is done in the `server setup` endpoint. This is why we cannot see a similar performance gain in the server-side registration and that the first three endpoints are relatively similar in terms of time consumption. Finally, KHAPE without OPRF is around 207 µs faster than KHAPE with OPRF — the equivalent of about three exponentiations — and 126 µs faster than OPAQUE, which makes it the fastest protocol.



Authentication

For the authentication, OPAQUE and KHAPE with OPRF have rather different endpoint performances but the overall performance of each protocol is almost equal with a difference of around seven µs. This is because a lot of similar operations are computed on both protocol but they are executed in different endpoints. Similar to the registration, KHAPE without OPRF is the fastest protocol. It is faster by 197 µs on OPAQUE and by 205 µs on KHAPE with OPRF. Even if it is the fastest in both registration and authentication, we believe that the security sacrifice is not worth the performance gain and therefore we recommend using OPAQUE or KHAPE with OPRF which is secure against pre-computation attacks (see Section 3.3.5).



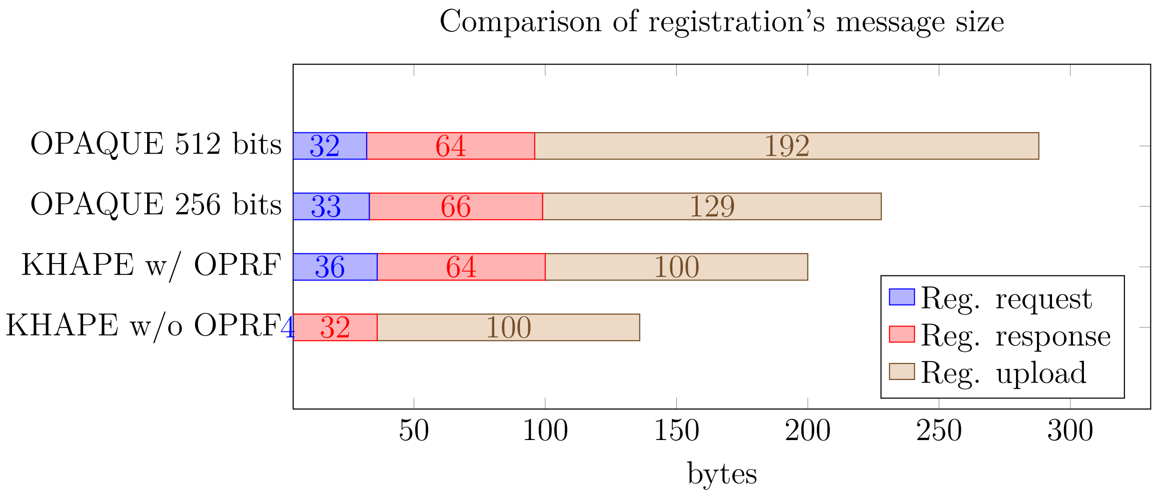
6.6 Message size

All the above benchmarks are only comparing computational performances by measuring the time consumption. In a real-world scenario, network delay during transmissions of messages between the client and the server also has to be considered. Network delay benchmark is difficult to evaluate and not very meaningful because it depends too much on the network infrastructure between the two parties. Nevertheless, this section compares the message size and message number for multiple OPAQUE and KHAPE configurations. OPAQUE results are taken from the standard draft test vectors and verified with the OPAQUE rust library [26]. In these tests vectors, the user identifier is four bytes long. The same length will be used for KHAPE's user id. It is interesting to note that in OPAQUE, the user id is not transmitted with the protocol's messages. It is the responsibility of the application to handle it. In KHAPE, the user id is included with the protocol's messages and accessible to the application.

For OPAQUE, the message size depends on the primitives used (cipher suite). In particular, the choice of the hashing function, MAC and KDF has a considerable impact since they can produce output of different length depending on the primitive used. For the benchmark, we will use a cipher suite with 256-bit output for hashing function, MAC and KDF because KHAPE also use this output size for these functions. A comparison is also made with a 512-bit output cipher suite. Note that the 256-bit cipher suites use a P256 curve where the points are represented on 33 bytes. The 512-bit cipher suite use Ristretto255 which is represented on 32 bytes.

Registration

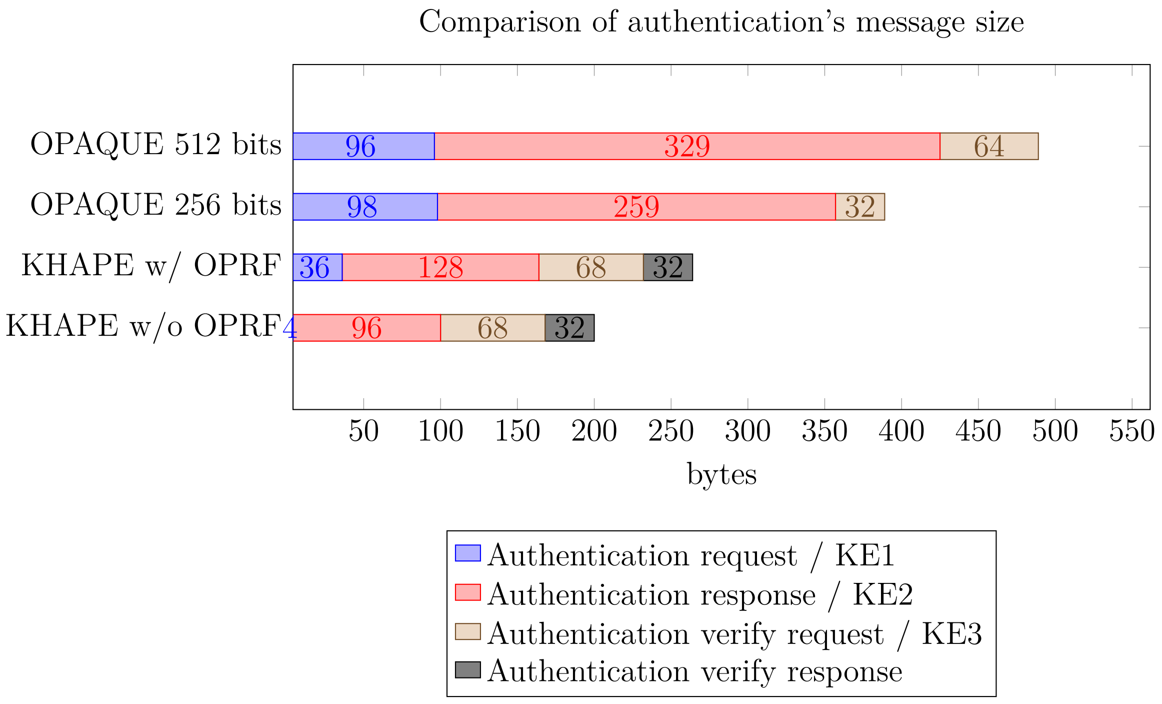
Firstly, comparing KHAPE with and without OPRF, we see that the first two messages contains 32 more bytes with OPRF. This is simply the size of the representation of a curve point used for OPRF. This difference is exactly the same for the authentication. Secondly, comparing KHAPE and OPAQUE 256-bit cipher suite, we notice that the last message is larger with OPAQUE. This is because OPAQUE encrypt the envelope before transmission with a one-time pad key. This key is derived from a static shared secret which is included in this registration upload message (see Section 2.2.3). The difference of size is due to this additional key sent with OPAQUE. Finally, comparing the two OPAQUE's cipher suites, we see that the first two messages contain curve points since the 256-bit cipher suite represents curve points on 33 bytes instead of 32. The last message contains two hash-derived output. Their size is doubled with the 512-bit cipher suites.



Authentication

For the registration, we can see a relatively large difference between KHAPE and OPAQUE 256-bit cipher suites in particular in the second message. The first message already is about three times longer with OPAQUE. This is because in addition to sending the OPRF blinded element like KHAPE do, OPAQUE also send the client's ephemeral public key and a nonce. KHAPE sends the client's ephemeral public key in the third message. In contrary to the server nonce that is used as a context in key derivation, the client nonce has no use in the OPAQUE's internet standard draft. For the second message, both OPAQUE and KHAPE sends the encrypted envelope, the OPRF evaluation result and the server's ephemeral public key. Due to its different envelope encryption (see Section 2.2.3), OPAQUE also sends a masking nonce and the server's public key (which is not included in the envelope in contrary to KHAPE). OPAQUE also sends a server nonce used in key derivation and the first verification tag. For the third message, OPAQUE sends the last verification tag where KHAPE only sends the first verification tag with the server ephemeral key. Only KHAPE sends a fifth message containing the last verification tag. Even though KHAPE's overall message size is smaller, the fact that it has one more message KHAPE's overall message size is smaller but it has one more message than OPAQUE. Having more message is generally more time consuming than having larger messages with all the overhead of a single message.

Similar to the authentication, KHAPE without OPRF has 32 bytes less for the first two messages than KHAPE with OPRF.



7 | Conclusion

This chapter concludes this report with a summary of the work done, and what can still be done.

7.1 Password security

PAKE protocol provides the highest level of security for password-based authentication, and KHAPE provide the largest number of security guarantees among PAKE protocol. This makes KHAPE the most secure way to perform authentication with passwords. However, passwords are still insecure by their very nature of being low-entropy secrets. Password-based authentication should ideally be replaced by more secure protocol such as WebAuthn.

7.2 Final result

Looking back at the goals defined in the specification, almost all of them have been achieved. The comparison between the main PAKEs was extensive. KHAPE has been chosen to be implemented, providing the first ever implementation of this protocol. A working prototype of an online password manager has been implemented for the use case. And a performance test has been added to evaluate the efficiency of the developed library. In this regard, the developed library achieve almost similar performances than the OPAQUE's library with a slightly higher security level which is a great success. The only thing that has not been done — due to lack of time — is the PAKE history. This section was supposed to provide an overview of the state of PAKE protocols — from the most commons to the less known — in a chronological order. In consequence, this report rarely mention any other PAKE protocols that is not one of the main protocols (EKE, SRP, OPAQUE and KHAPE). However, the explanations and comparison of the main PAKE protocols are much more significant than initially planned and provide — in part — the overview that the history should have provided, at least for the main protocols.

7.3 Future work

The KHAPE library has been implemented and is functional but multiple interesting concept has not been considered or implemented by lack of time.

- Protection against timing attacks,
- Discharge passwords, and other sensitive value from memory directly after use,
- Password reset feature,
- Support for WebAssembly build (to be used directly in web browsers),
- Refactor the library to be generic and give the choice of the cryptographic primitives to the library's users,
- Performance comparison between 3DH and HMQV,
- Adapt the online password manager client for web browsers

7.4 Personal conclusion

I had much pleasure to do this work and to implement KHAPE and I will continue to improve it in order to propose a public library. Reading a large number of cryptographic paper was new to me but it was very interesting to see how the cryptographic literature evolved with time. In terms of difficulties, KHAPE's non-committing encryption was hard to understand and implement for me. It took me quite some time to understand the concept and how it works and even more time to find a way to implement it in the library. Other than that, I didn't face any big difficulties.

Bibliography

[1] IEEE standard specification for password-based public-key cryptographic techniques. *IEEE Std 1363.2-2008*, pages 1–140, 2009.

[2] Information technology — security techniques — key management — part 4: Mechanisms based on weak secrets. *ISO/IEC*, 11770-4, 2017.

[3] Algorithms, key size and protocols report. *ECRYPT-CSA*, H2020-ICT-2014 — Project 645421, 2018.

[4] Bitwarden security white paper. `https://bitwarden.com/images/resources/security-white-paper-download.pdf`, 2020.

[5] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 139–155. Springer, 2000.

[6] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society, 1992.

[7] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In *CCS*, pages 244–250. ACM, 1993.

[8] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *CCS*, pages 967–980. ACM, 2013.

[9] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *EuroS&P*, pages 292–302. IEEE, 2016.

[10] Daniel Bourdrez, Dr. Hugo Krawczyk, Kevin Lewi, and Christopher A. Wood. The OPAQUE Asymmetric PAKE Protocol. Internet-Draft draft-irtf-cfrg-opaque-07, Internet Engineering Task Force, 2021. Work in Progress.

[11] Tatiana Bradley. OPAQUE: the best passwords never leave your device. `https://blog.cloudflare.com/opaque-oblivious-passwords/`, 2020.

[12] Julien Bringer, Hervé Chabanne, and Thomas Icart. Password based key exchange with hidden elliptic curve public parameters. *IACR Cryptol. ePrint Arch.*, page 468, 2009.

[13] Jean-Sébastien Coron, Jacques Patarin, and Yannick Seurin. The random oracle model and the ideal cipher model are equivalent. In *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2008.

[14] Alex Davidson, Armando Faz-Hernández, Nick Sullivan, and Christopher A. Wood. Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups. Internet-Draft draft-irtf-cfrg-voprf-08, Internet Engineering Task Force, 2021. Work in Progress.

[15] Alexandre Duc. Cryptographie avancée appliquée [advanced applied cryptography]. University Lecture at HEIG-VD – Haute École d'Ingénierie et de Gestion du Canton de Vaud, 2021.

[16] Rick Fillion. Developers: How we use srp, and you can too. `https://blog.1password.com/developers-how-we-use-srp-and-you-can-too/`, 2018.

[17] Craig Gentry, Philip D. MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2006.

[18] Matthew Green. Let's talk about PAKE. `https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/`, 2018.

[19] Matthew Green. Should you use SRP ? `https://blog.cryptographyengineering.com/should-you-use-srp/`, 2018.

[20] Laurent Grémy, Guillaume Heilles, and Nicolas Surbayrole. Security audit of dalek libraries. `https://blog.quarkslab.com/security-audit-of-dalek-libraries.html`, 2019.

[21] Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. KHAPE: asymmetric PAKE from key-hiding key exchange. In *CRYPTO 2021*, volume 12828 of *Lecture Notes in Computer Science*, pages 701–730. Springer, 2021.

[22] Thomas Holenstein, Robin Künzler, and Stefano Tessaro. The equivalence of the random oracle model and the ideal cipher model, revisited. In *STOC*, pages 89–98. ACM, 2011.

[23] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT 2018*, volume 10822 of *Lecture Notes in Computer Science*, pages 456–486. Springer, 2018.

[24] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[25] Hugo Krawczyk and Pasi Eronen. Hmac-based extract-and-expand key derivation function (HKDF). *RFC*, 5869:1–14, 2010.

[26] Kevin Lewi and François Garillot. OPAQUE-KE rust library. `https://github.com/novifinancial/opaque-ke`, 2021.

[27] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[28] Colin Percival and Simon Josefsson. The scrypt password-based key derivation function. *RFC*, 7914:1–16, 2016.

[29] Yaron Sheffer, Glen Zorn, Hannes Tschofenig, and Scott R. Fluhrer. An EAP authentication method based on the encrypted key exchange (EKE) protocol. *RFC*, 6124:1–33, 2011.

[30] Alan T. Sherman, Erin Lanus, Moses Liskov, Edward Zieglar, Richard Chang, Enis Golaszewski, Ryan Wnuk-Fink, Cyrus J. Bonyadi, Mario Yaksetig, and Ian Blumenfeld. Formal methods analysis of the secure remote password protocol. In *Logic, Language, and Security*, volume 12300 of *Lecture Notes in Computer Science*, pages 103–126. Springer, 2020.

[31] David Taylor, Thomas Wu, Nikos Mavrogiannopoulos, and Trevor Perrin. Using the secure remote password (SRP) protocol for TLS authentication. *RFC*, 5054:1–24, 2007.

[32] Mehdi Tibouchi. Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In *Financial Cryptography*, volume 8437 of *Lecture Notes in Computer Science*, pages 139–156. Springer, 2014.

[33] Thomas Wu. The SRP authentication and key exchange system. *RFC*, 2945:1–8, 2000.

[34] Thomas Wu. Telnet authentication: SRP. *RFC*, 2944:1–7, 2000.

[35] Thomas Wu. SRP-6: Improvements and refinements to the secure remote password protocol. *http://srp.stanford.edu/srp6.ps*, 2002.

[36] Thomas D. Wu. The secure remote password protocol. In *NDSS*. The Internet Society, 1998.

[37] Muxiang Zhang. Breaking an improved password authenticated key exchange protocol for imbalanced wireless networks. *IEEE Commun. Lett.*, 9(3):276–278, 2005.

List of Figures