HEIG VD 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, December 19, 2021

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

Specification **iii**

1 Introduction **1**

1.1 Problematic . . . 1

1.2 My contribution . . . 3

2 State of the art **5**

2.1 Notation . . . 5

2.2 History of PAKEs . . . 5

2.2.1 Symmetric PAKE . . . 6

2.2.2 Asymmetric PAKE . . . 6

2.3 Main PAKEs . . . 6

2.3.1 EKE . . . 6

2.3.2 SRP . . . 8

2.3.3 OPAQUE . . . 10

2.3.4 KHAPE . . . 13

2.4 Comparing main solutions . . . 15

2.4.1 Details . . . 15

2.4.2 Table . . . 19

3 KHAPE **21**

3.1 Choice of implementing KHAPE . . . 21

3.2 Generic algorithm . . . 21

3.3 Design choices 21
3.3.1 Client-side register 22
3.3.2 Key Exchange 22
3.3.3 Encryption scheme 22
3.3.4 Group $\mathbb{G}$ 23
3.3.5 OPRF 24
3.3.6 Slow Hash 24
3.3.7 Key derivation 24
3.3.7.1 Encryption key and export key 25
3.3.7.2 Output key and key verification 25
3.3.8 Session during authentication or registration ? 26
3.3.9 Hash 26
3.3.10 HashToGroup 26
3.3.11 PRF 26
3.3.12 GenerateRandomNumber 26
3.4 Limitations 26
3.4.1 Client 27
3.4.2 Server 27
3.5 Security consideration 27
3.5.1 Difference with OPAQUE 27
3.5.2 Input validation 28
3.5.3 SlowHash 28
3.5.4 Session key usage 28
3.6 Precise algorithm 28
3.7 (Threat model) 28

4 Implementation **33**
4.1 Parameters 33
4.2 Exchanges 33

4.2.1 Register 33
4.2.2 Login 34
4.3 Function definition 34
4.3.1 Client 34
4.3.2 Server 34
4.4 Messages structures 34
4.5 Library choices 36
4.6 API 38
4.6.1 Client 38
4.6.2 Server 38
4.6.3 Messages 38
4.6.4 Common 38
4.6.5 Client register 38
4.6.6 Server register 38
4.6.7 Client login 38
4.6.8 Server login 39
4.6.9 Structure 39
4.7 Dependencies/Libraries choices 39
4.8 Code structure 39
4.9 Interesting functions 39
4.9.1 Discharge password from RAM directly after use 39
4.9.2 (Timing attack mitigation) 39

5 Use case 41
5.1 Online password manager 41
5.2 Other use case 41
5.3 Design 42

6 Results 43
6.1 KHAPE endpoints 43
6.1.1 Ideal Cipher vs AES 44

6.1.2 With OPRF vs without OPRF . . . 44
6.1.3 With SlowHash vs without SlowHash . . . 45
6.2 Components benchmark . . . 46
6.2.1 Encryption (vs AES) . . . 46
6.2.2 TripleDH . . . 47
6.2.3 Key generation . . . 47
6.3 OPAQUE endpoints . . . 47
6.4 OPAQUE vs KHAPE . . . 49
6.5 Message size . . . 49

7 Conclusion **51**
7.1 Future work . . . 51
7.2 Ideas not implemented . . . 51

Bibliography **53**

1 Introduction

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

1.1 Problematic

How to authenticate a user ? When a user want to connect itself to a online service, he send 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 send 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 the specific user. This means that the server has to knows and store this sensible value before authentication — generally during register.

Currently on the vast majority of websites and softwares, passwords are used 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 get compromised, the attacker immediately obtain access to all passwords since the server store the passwords. This means that the adversary can impersonate every user.

To avoid this scenario, numerous techniques have been developed. Mainly, adding salt, adding pepper — a secret salt — and using memory-hard password hashing function such as Scrypt [19] or Argon2 [7].

These techniques improve the security of storing password but they doesn't 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" [8]. Now-a-day, a majority of passwords use require that the password is sent in cleartext.

Even if the channel between the client and the server is appropriately secured, generally with TLS (PKI attack, cert miss-configuration, etc.), and even if on the server-side every secure password storing techniques are 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 forgot. While the password is processed in clear, it reside on the memory. It use a shared bus between the CPU and the memory. Hardware attacks are less likely to occur but are no less severe (remember Spectre and Meltdown).

In a ideal world, the server should never see the user's password in cleartext at all.

Get rid of password. In summary, password 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.

These solutions are a good replacement to passwords but they require a deep change. It will take time for them to grow mature and impose themself as industry standard. This is also because password are so ubiquitous due in part to the ease of implementation and the familiarity for the users. If we cannot get rid of passwords for now, we need a way to make it "as secure as possible while they persist".

This is where PAKE become interesting. It allow password-based authentication without the password 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 party knows 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.

aPAKE guarantee that the client's password is protected because it never leave the client's machine in cleartext. It is done by doing a key exchange between the client and the server. It allow mutual authentication in a client-server scenarios without requiring a PKI (except for the initial registration).

"A secure aPAKE should provide the best possible security for a password protocol" [17].

And should only be vulnerable to inevitable attacks such as online guess or offline dictionary attacks if server's data get leaked.

Why PAKEs have almost no adoption ? Despite existing for nearly 3 decades and providing better security guanrantees than traditional authentication method, PAKEs have almost no adoption. So why are they so rare in the industry now-a-day ?

Firstly, for web site, it's easier to setup a password form and handle 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 was either patented, got broken or both. It probably hurted the reputation and adoption of PAKEs. Another factor is the insufficiency of well-implemented PAKE library in some programming language which make them difficult to use.

One exception to that is SRP, the most used PAKE protocol in the world. It is a TLS ciphersuite, is implemented in OpenSSL and used in Apple's iCloud Key Vault. Even though it has far more adoption than other PAKEs, is not the ideal PAKE.

Today, new generation of PAKE are better and provide more security guarantees. Efforts are made to make PAKE a standard for password authentication.

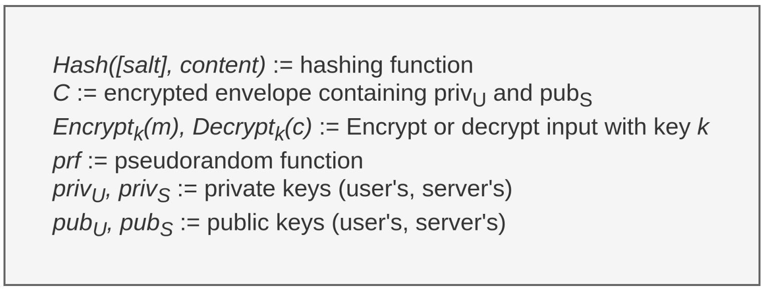
1.2 My contribution

2 State of the art

This chapter aim to provide a detailed view of the current PAKE landscape from the oldest to the most recent construction.

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 History of PAKEs

EKE (1992).

EKE variants (PAK, PPK, PAK-X).

OKE.

SNAPI.

PEKEP.

2.2.1 Symmetric PAKE

2.2.2 Asymmetric PAKE

SRP (1998).

OPAQUE (2018).

KHAPE (2021).

2.3 Main PAKEs

This section describe in more details four fundamental PAKE construction. EKE as it is the first ever PAKE. SRP because it's 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 Srong aPAKE KHAPE because it is very recent and provide slightly better security guarantees than OPAQUE in certain conditions.

2.3.1 EKE

Introduction. EKE (for Encrypted Key Exchange) was proposed in 1992 by Bellovin and Merritt [5] 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 [27].

Overall, EKE got broken and therefore should not be used.

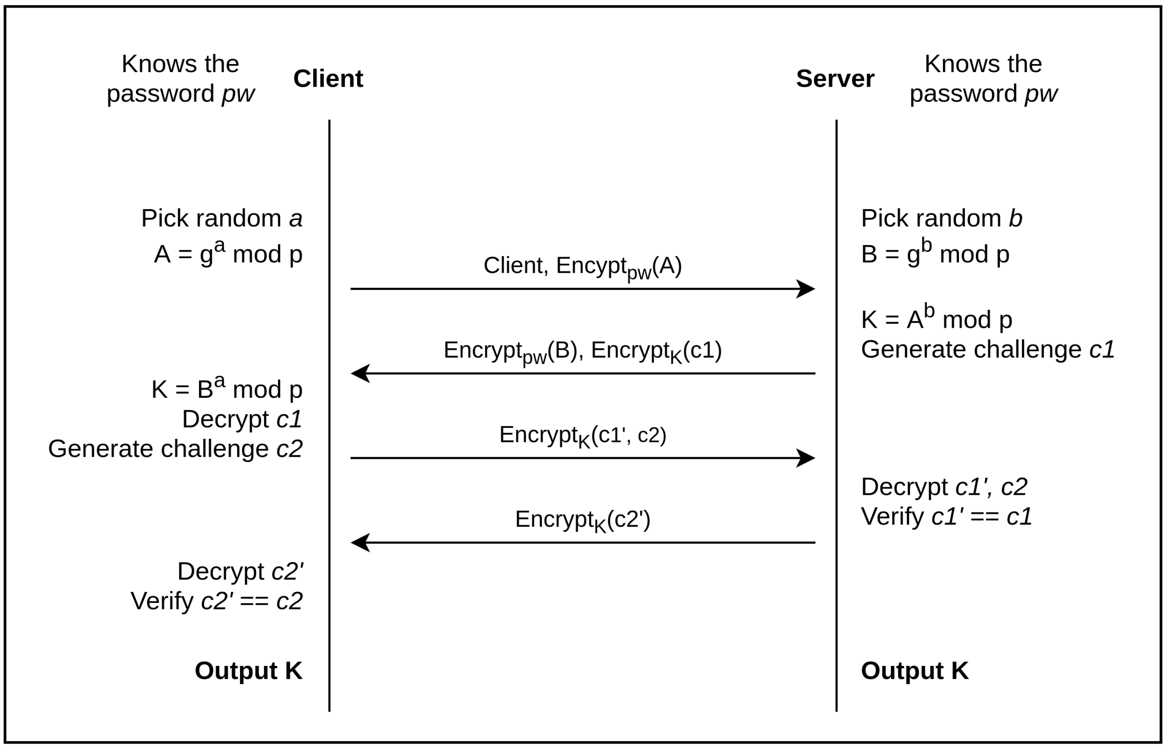


Figure 2.2: Login process with EKE (DH-EKE) protocol.

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's 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's 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 doesn't 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.

2.3.2 SRP

Introduction. SRP [26, 25] (for Secure Remote Password) was proposed in 1998 and is the most widely-implemented PAKE protocol in the world [13]. It's largely used in iCloud Key Vault — which could make it one of the most widely-used cryptographic protocols [13] considering the number of active Apple devices worldwide — and in 1Password's password manager. It is well standardized and has numerous implementation in different programming languages. It is in fact a TLS ciphersuite [22], implemented in OpenSSL.

This success is partly due to the SRP's creators will to avoid patent — unlike most of the PAKE of its time — but also to avoid export restriction imposed by US law by not using any encryption schema [21]. 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 doesn't 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 make SRP easy to implement for developers and transparent for the user.

One of the main strength of SRP is that the server doesn't store the cleartext password or the hashed password. Instead it store 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 send 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. "SRP protocol design is completely bonkers" The protocol mix addition and multiplication in calculation. Using both operations require a ring rather that a cyclic group. This requirement make it impossible to easily transfer the integer-based algorithm to elliptic curve.

This requirement, also make 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 $v + g^b \mod q$" [21].

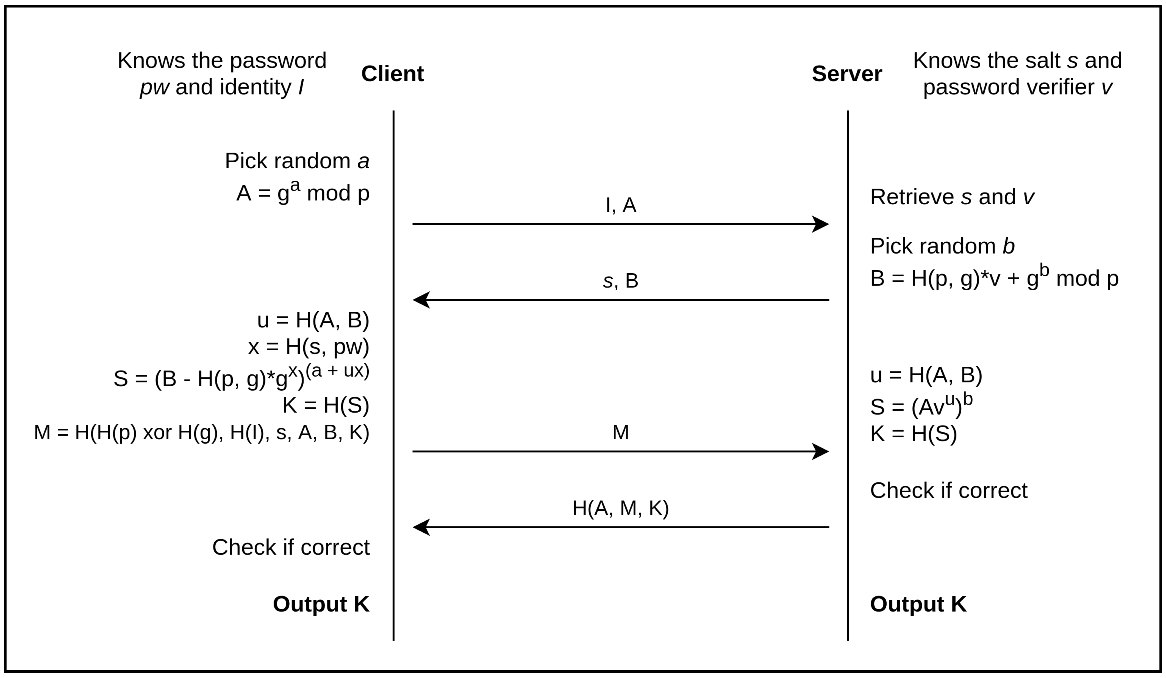


Figure 2.3: Login process with SRP-6a protocol.

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 compute A.
2. Client sends A and its identity I (username) to the server.
3. Server retrieves user's 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 pick a random b and compute B.
6. Server sends s and B to the client.
7. Client and server both compute u, S and K with their own values.

8. They finish with a mutual key confirmation where the client sends his proof of K first.
9. If the server find that the user's proof is incorrect, he stop the exchange and doesn't sends 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. Registration process is not covered in SRP papers. Client 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 it's register request — through a secure channel — and the server generate a salt and compute the verifier (Easier to implement 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>`

2.3.3 OPAQUE

Introduction. Jarecki et al. [16]. 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.

These functions allow for each party, namely the client and the server, to input a secret value and then the client can use the output as a key. Neither party can learn the other party's secret, and the server cannot learn the output of the function.

Overall, the OPAQUE protocol allows to secure authentication from the simplest applications to the most sensitive ones.

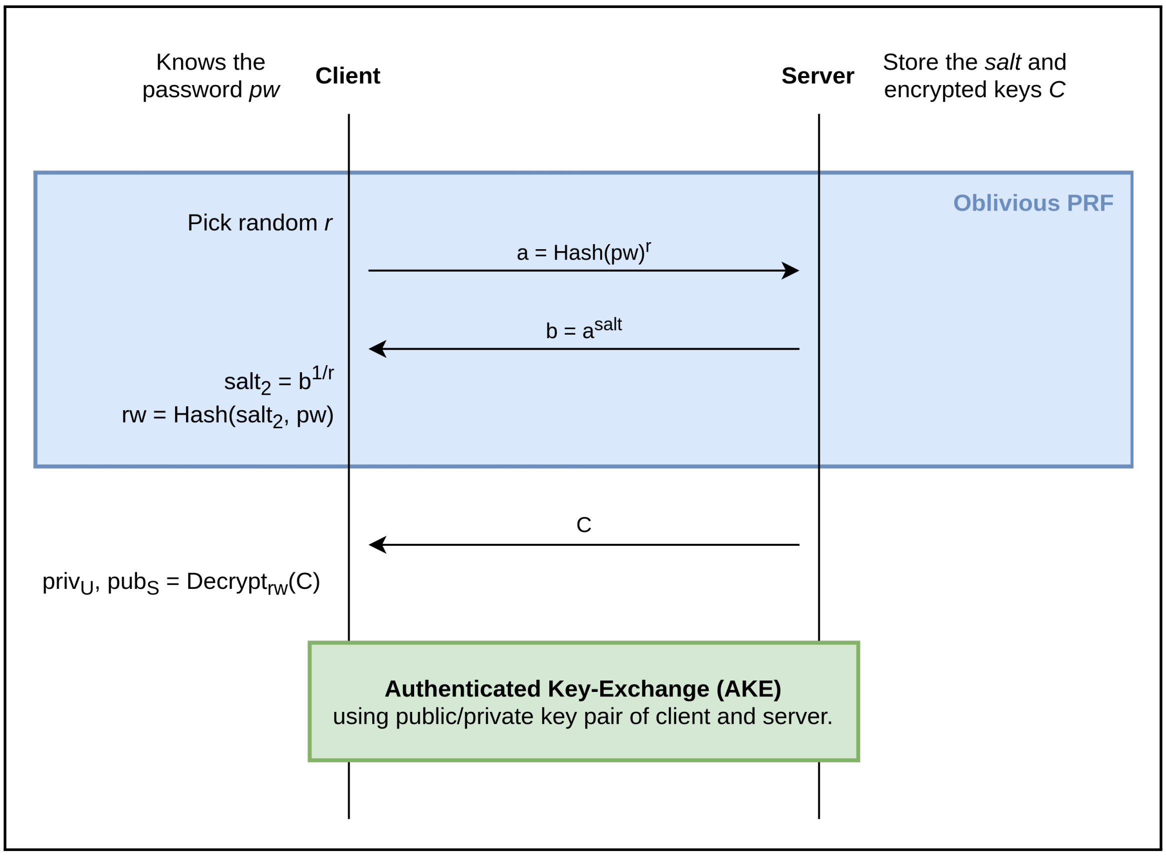

Knows the password *pw*
Client
Server
Store the *salt* and encrypted keys *C*
Oblivious PRF
Pick random *r*
a = Hash(pw)^r
b = a^{salt}
salt$_2$ = b$^{1/r}$
rw = Hash(salt$_2$, pw)
C
priv$_U$, pub$_S$ = Decrypt$_{rw}$(C)
Authenticated Key-Exchange (AKE)
using public/private key pair of client and server.

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

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.
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.

[16] also note 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 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.

2.3.4 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) [15] 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 [16] 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 server-side threshold implementation that doesn't require any change from the client. 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 it's private key and server's public key on the server, encrypted by his password.

Without using an OPRF, an adversary could try to initiate a authentication request with the server with another user's id to obtain it's credentials ciphertext.

Without using an OPRF, 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 find a match with the server's public key that he recorded sooner. To avoid this kind of attacks, KHAPE use two mains mechanisms: Non-commiting encryption and key-hiding AKE.

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

Key-hiding AKE. The key-hiding feature make 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.

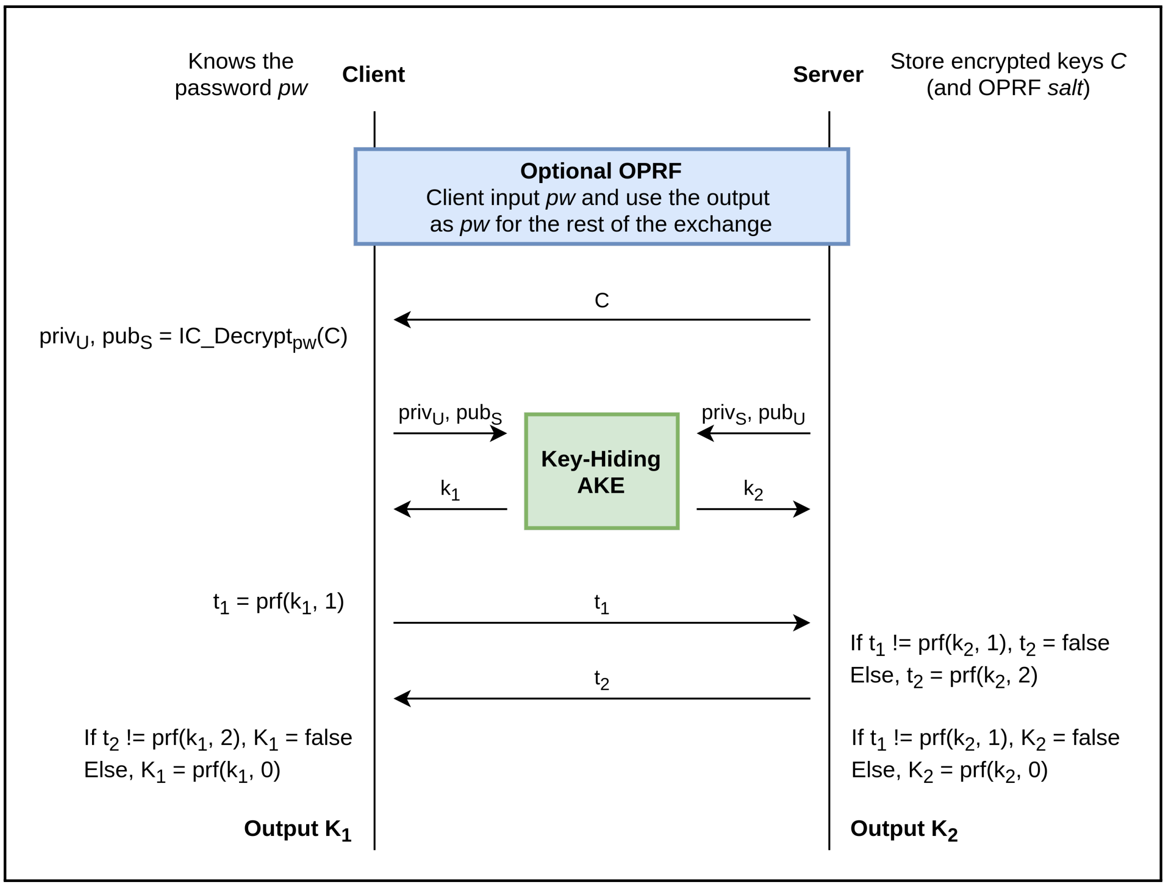

Knows the password *pw*
Client
Server
Store encrypted keys *C* (and OPRF *salt*)
Optional OPRF
Client input *pw* and use the output as *pw* for the rest of the exchange
C
$priv_U$, pub_S = $IC\_Decrypt_{pw}(C)$
$priv_U$, pub_S
$priv_S$, pub_U
Key-Hiding AKE
k_1
k_2
$t_1 = prf(k_1, 1)$
t_1
If t_1 != $prf(k_2, 1)$, t_2 = false
Else, $t_2 = prf(k_2, 2)$
t_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. 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 [16]. 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 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 that is addressed in 2.3.3. 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, the paper proposes a client-side registration process.

2.4 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 2.4.1.

2.4.1 Details

1. Server doesn't process passwords in cleartext. This is the main security property of asymmetric PAKE [12]. Server doesn't 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 criteria 1, it's not. Criterion 1 is about password processing, but this criterion is about password transmission. Transmission and processing of passwords are vulnerable to different attacks vectors. The server doesn't receive passwords in cleartext which avoid any miss-handling vulnerabilities such as logging or caching cleartext passwords on the server.

COMMENT: May be required during register depending on the implementation (See Section 2.3.3).

3. Secure against pre-computation attacks. This is the main security property of Strong aPAKE [16]. The server doesn't 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*. So this protection force the attacker to perform an offline dictionary attack after successful server compromise.

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 doesn't.

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 transmissions between client and server doesn't 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 [19] or Argon2 [7] 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 [6] 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 [11].

8. Built-in mechanism to store client's secrets on the server. Securely store client's sensible 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 server 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 an highly sensitive application. OPRF transparently provide this functionality where each server add 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 use 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" [15]. 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 adoption of an construction.

12. Security proof. COMMENT: EKE only provide informal security analysis [4]

SRP provide no valuable security proof [10, 14]. It only prove that it can stands up to passive attacks, which is not enough for an authentication protocol. A proof against active attacks would be welcomed for a such widely-used protocol.

13. Easily adaptable to elliptic curves. Elliptic curves cryptography allow to greatly reduce the size of asymmetric key. This is crucial in term of performance because keys size recommendation are always growing to ensure security and asymmetrical key are getting giant and difficult to manage — in particular for key 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]. In comparison, with elliptic curves, only 512 bits are recommended to archive similar security level.

COMMENT: In SRP, $\mathbb{Z}_p$ is used as a field, not a group. Therefore, SRP cannot be easily adapted to elliptic curves [10].

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 make it impossible to implement it on elliptic curves [9].

14. Number of messages. more messages means "increasing latency and load on the network"

15. Number of exponentiations.

16. Computational cost compared to a KE (see [15] presentation).

17. Communication size.

18. Server-side storage size.

19. Patented.

20. Year published.

21. Got broken.

2.4.2 Table

For value where there is an "*", please refer to the appropriate comment in Section 2.4.1 for precision.

#	Criteria	EKE	SRP	OPAQUE	KHAPE
1	Server doesn't 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, if using OPRF
4	Forward secrecy	Yes*	Yes	Yes	Yes
5	Mutual authentication	Yes	Yes	Yes	Yes
6	PKI-free	Yes, except during register	Yes, except during register	Yes, except during register	Yes, except during register
7	User-side password hardening	No*	Yes*	Yes	Yes, if using 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, if using 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 [20]	3 RFC [24, 23, 22], 1 ISO [2], 1 IEEE [1]	Internet standard draft [17]	CRYPTO 2021 Paper [15]
12	Security proof	No*	No valuable security proof	Yes, in the random oracle model	Yes, in the ideal cipher model

#	**Criteria**	**EKE**	**SRP**	**OPAQUE**	**KHAPE**
13	Easily adaptable to elliptic curves	No*	No*	Yes	Yes
14	Number of messages	4 ?	4 ?	3	4 (3 if client initiate)
15	Number of exponentiations	4 ?	4 ?	3 or 4 ?	2 + 1 hash-to-curve
16	Computational cost compared to a KE (see [15] presentation)	1x	TBD	2x	1x without OPRF, 2x with OPRF
17	Communication size	TBD	TBD	TBD	TBD
18	Server-side storage size	TBD	TBD	TBD	TBD
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 [10]	No	No

3 KHAPE

This chapter explain the details of the KHAPE protocol in term 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 and/or standard 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, KHAPE paper was published only 6 month ago at the time of the writing. To my knowledge, there is currently no public implementation of KHAPE and this is why I will be implementing it.

3.2 Generic algorithm

This sections details a generic KHAPE protocol. Algorithm 1 and 2 show the pseudocode of a login process from both client-side and server-side. Algorithm 3 and 4 show the register process.

3.3 Design choices

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

3.3.1 Client-side register

Both client-side and server-side registration are possible but in order to use the aPAKE benefit to the fullest, it is preferable to handle the registration on the client. This allow to keep the main goal of aPAKEs: the server NEVER see the user's password. For more details, see Section 2.3.3.

3.3.2 Key Exchange

PAKE protocol compute 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 unauthenticated KE protocol like plain Diffie-Hellman.

In AKE protocols, each party has two asymmetric keys pair. 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 require that the underlying key exchange protocol is a "key-hiding AKE" (See Section 2.3.4 for more details about this construction). [15] shows that HMQV, 3DH and SKEME are key-hiding AKE.

SKEME is a AKE based on key encryption mechanism where HMQV and 3DH are Diffie-Hellman based AKE. [16] and [15] shows concrete instantiation of their protocol using HMQV. These instantiations are easily adapted to other Diffie-Hellman based AKE like 3DH.

HMQV :

$d_c \leftarrow$ Hash'(sid, C, S, 1, X)
$e_c \leftarrow$ Hash'(sid, C, S, 2, Y)
$o_c \leftarrow (Y \cdot B^{e_c})^{x+d_c \cdot a}$

3DH :

$o_c \leftarrow B^x||Y^a||Y^x$

Between 3DH and HMQV, the latter is more efficient but it is patented. This is why 3DH is used for the implementation.

3.3.3 Encryption scheme

KHAPE require that the encryption scheme is non-committing (See Section 2.3.4).

Non-committing encryption is archived by combining ideal cipher and curve encoding.

Ideal cipher. As its name may suggest, the ideal cipher model is an idealised model used to prove the security of cryptographical systems.

In practice, an ideal cipher is not implementable but it's 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 [?] which is an update from [?] that got broken. The encryption scheme consist of 14 rounds of Feistel where function $F_i = H(key|i|input)$ [1].

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 256 bits which is the size of a single group element.

In term of performance, this construction is not very efficient but its one of the only found that is easily instantiatiable.

Curve encoding. The cleartext group elements that will be encrypted need to be encoded as a bitstring that is indistiguishable from random. This has the consequence that if plaintext is encrypted with password pw and then decrypted with a different password pw', the result is a random valid element.

This is done by implementing a quasi bijection from field elements to bitstrings. Elligator-squared and Elligator2 are implementation 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 suitable for Elligator2.

Authentication. Authenticated encryption is generally recommended for encryption. It is interesting to note that this encryption scheme must not be authenticated.

In contrary to the OPAQUE Paper, the OPAQUE standard draft doesn't encrypt the client's private key in the envelope. Instead, the envelope store a nonce of 32 bytes and an authentication tag. The envelope is not encrypted. The nonce is used — with the randomized password — to compute a seed, which is then used to derive the client's private and public keys. The authentication tag stored in the envelope is used to verify the derived public key.

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 curves group.

[1]Thanks to Prof. Alexandre Duc for providing this construction

For performances reasons, elliptic curves are used for the implementation (See Section 2.4.1). 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 KHAPE implementation, curve25519 is used. It is a safe elliptic curve that provide the sufficient security level and is well suited for implementing Elligator2 on it. Group element are represented on 32 bytes.

3.3.5 OPRF

In KHAPE, the OPRF is optional — in contrary to OPAQUE — but allow to make it a strong aPAKE resistant against pre-computation attacks. This is a major improvement in term 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 [**?**] that present multiple ciphersuites for OPRF and different variants. The OPRF used for the KHAPE implementation is based on this standard draft using the ristretto255-SHA-512 ciphersuite and the standard non-verifiable variant.

3.3.6 Slow Hash

It is possible and recommended to use a memory-hard hashing function on the password to make it slow and expensive to compute. This make it more costly for an adversary to compute hashs.

For the implementation, Argon2id [7] is used because it is a recent memory-hard hashing function with a simple construction and it has better security analysis than others resource-heavy hashing function like scrypt, bcrypt and PBKDF2 [10]. The Argon2id variant is used as it provide resistance against both GPU attacks and side-channel attacks — and is the recommended version.

For OPAQUE, [17] 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.

3.3.7 Key derivation

This section describe the process of derivating keys from existing secrets. The design is heavily inspired by OPAQUE's standard draft [17] 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 [18]. It is well suited for expanding key from existing secret and is already used in OPAQUE rust's implementation [?]. It follows the "Extract-then-Expand" paradigm where these two functions are used with the following API :

- Extract(*salt*, *ikm*): Extract a fixed length pseudorandom 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 pseudorandom 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 an hashing function. The underlying hashing function used 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

OPRF output — or 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's passwords for an online password manager.

The following pseudo-code shows how these keys are derived.

```
**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 3 values are computed at the same time and stored until needed.

The following pseudo-code shows how these keys are derived.

```
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 $t_2$
```

3.3.8 Session during authentication or registration ?

3.3.9 Hash

Sha3-256

3.3.10 HashToGroup

Operation of hashing to the group. Since we use elliptic curves, this operation is an hash-to-curve.

3.3.11 PRF

HKDF

3.3.12 GenerateRandomNumber

Voir CAA

3.4 Limitations

In opposition with classical authentication method where the client only send an authentication request and receive a response. With KHAPE, the registration and the authentication processes require multiple messages (round-trip) between the client and the server. This add new difficulties for both side that need to be considered.

3.4.1 Client

Necessity to send 2 messages. If the client is a web browser, he cannot only send a login form to the server. Behind the scenes request has to be made to compute the 2 round-trip. This means that some javascript has to be implemented but this was known from the beginning that PAKE protocol require a greater implication of the client (code-wise).

3.4.2 Server

Pre register secret, ephemeral key, session The server communicate with multiple client at the same time so when a request come, he needs to know from which client/session it is from 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.

During authentication, the server needs to store his ephemeral private key for the second round-trip. He can store this value in the user's file entry or in a separated session file.

He can : - Store this value in the file entry associated to the user - Store this value in a session file entry (this require that client generate a session id and sends it with every request) (can be used to store session key later) - Encrypt the value and send it to the client which send it back (like JWT) In every case, the client has to resend the uid in the 2nd round-trip request

The second solution will be used. Allow multiple session

During register, the server generate his keys (private and public) and the secret salt. He sends the public key but the private key and the secret salt has to remain secret.

He can : - Pre register the client in a incomplete file entry. - file[uid] = <secret_salt, b, INCOMPLET> - file[uid] = <e, b, A, secret_salt, COMPLET> - Solution 2 and 3 from auth

The first solution will be used

3.5 Security consideration

3.5.1 Difference with OPAQUE

Verification must be initialized by the client, this mean that protocol in completed with 4 message instead of 3 for OPAQUE. Still use encryption (ideal cipher) where the OPAQUE standard draft doesn't use encryption anymore. Use key-hiding AKE (3DH and HMQV are key-hiding AKE so this doesn't change anything).

3.5.2 Input validation

"We assume that parties verify public keys and ephemeral DH values, resp. B; Y for P1 and A; X for P2, as group G elements."

3.5.3 SlowHash

Optionaly, use Argon2, scrypt, etc. on the password before inputting it in the algo

3.5.4 Session key usage

3.6 Precise algorithm

3.7 (Threat model)

Algorithm 1 KHAPE : Authentication on the client (generic algorithm)

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

Algorithm 2 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 3 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 4 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

4.1 Parameters

Both client and 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 every 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 Strongly encouraged, improve security by a lot, doesn't take much time/resource.

Slow Hash Encouraged, improve security, take lot of time and resources

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

4.2 Exchanges

4.2.1 Register

```
(register_request, oprf_client_state) = client.register_start(password);
----------------register_request----------------->
(register_response, pre_register_secrets) = server.register_start(register_request);
<-----------------register_request----------------
register_finish = client.register_finish(register_response, oprf_client_state);
----------------register_finish----------------->
file_entry = server.register_finish(register_finish, pre_register_secrets)
                                                store file_entry
```

4.2.2 Login

```
(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) = client.auth_ke(auth_response, oprf_client_state)
----------------auth_verify_request---------------->
(auth_verify_response, server_output_key) = server.auth_finish(auth_verify_request,
<-----------------auth_verify_response----------------
client_output_key = client.auth_finish(auth_verify_response, ke_output)
```

4.3 Function definition

4.3.1 Client

- `register_start(&self, password: &[u8]) -> (RegisterRequest, ClientState)`
- `register_finish(&self, register_response: RegisterResponse, client_state: Clien`
- `auth_start(&self, password: &[u8]) -> (AuthRequest, ClientState)`
- `auth_ke(&self, auth_response: AuthResponse, client_state: ClientState) -> (Auth`
- `auth_finish(&self, auth_verify_response: AuthVerifyResponse, ke_output: KeyExch`

4.3.2 Server

- `register_start(&self, register_request: RegisterRequest) -> (RegisterResponse,`
- `register_finish(&self, register_finish: RegisterFinish, pre_register_secrets: P`
- `auth_start(&self, auth_request: AuthRequest, file_entry: &FileEntry) -> (AuthRe`
- `auth_finish(&self, auth_verify_request: AuthVerifyRequest, ephemeral_keys: Ephe`

4.4 Messages structures

```
pub struct RegisterRequest {
    pub uid: String,
    pub(crate) oprf_client_blind_result: Option<Vec<u8>>,
}

pub struct RegisterResponse {
    pub(crate) pub_b: PublicKey,
    pub(crate) oprf_server_evalute_result: Option<Vec<u8>>,
}

pub struct RegisterFinish {
    pub uid: String,
    pub(crate) encrypted_envelope: EncryptedEnvelope,
    pub(crate) pub_a: PublicKey
}


pub struct AuthRequest {
    pub uid: String,
    // pub sid: String, // TODO sid
    pub(crate) oprf_client_blind_result: Option<Vec<u8>>,
}


pub struct AuthResponse {
    pub(crate) encrypted_envelope: EncryptedEnvelope,
    pub(crate) pub_y: PublicKey,
    pub(crate) oprf_server_evalute_result: Option<Vec<u8>>,
}


pub struct AuthVerifyRequest {
    pub uid: String,
    // pub sid: String, // TODO sid
    pub(crate) client_verify_tag: VerifyTag,
    pub(crate) pub_x: PublicKey,
}


pub struct AuthVerifyResponse {
    pub(crate) server_verify_tag: Option<VerifyTag>,
}
```

4.5 Library choices

- `curve25519-dalek v3.2.0` : Pure rust implementation of group operations on Ristretto and Curve25519. Audited in 2019 [?]. Patched with PR [?] to implement elligator_decode
- `voprf v0.3` : Implementation of a verifiable OPRF based on the standard draft [].
- `sha3 v0.9` : Pure rust implementation of the SHA-3 (Keccak) hash function
- `hkdf v0.11` : Pure rust implementation of the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) generic over hash function
- `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 efficiently and generically
- `serde-big-array v0.3` : Big array helper for serde

```
[patch.crates-io]
curve25519-dalek = { path = "../07-curve25519-dalek-fork" }

[dependencies]
curve25519-dalek = { version = "3.2.0", features = ["serde"] }
voprf = "0.3.0"
rand = "0.8.4"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
#bincode = "1.3.3"

sha3 = "0.9.1"
hkdf = "0.11"
argon2 = "0.3.1"
serde-big-array = { version = "0.3.2", features = ["const-generics"] } # for ciphert
```

.

4.6 API

4.6.1 Client

4.6.2 Server

4.6.3 Messages

4.6.4 Common

- `generate_asymetric_key`

4.6.5 Client register

- `client_register_start(uid, pw): RegisterRequest`
- *Sends uid and h1. Receive B and h2*
- `client_register_finish(B, pw, h2) : e`
- *Sends e and A*

4.6.6 Server register

- *Receive uid and h1*
- `server_register_start(uid, h1): (b, B, salt, h2)`
- *Response B and h2*
- *Receive e and A*
- `server_register_finish(e, A, b, salt): file_entry`

4.6.7 Client login

- `client_auth_start(uid, pw): RegisterRequest`
- *Sends uid and h1. Receive e, Y and h2*
- `client_auth_ke(e, Y, h2, r, pw) : (k1, t1, X)`

- *Sends t1 and X. Receive t2*
- `client_auth_finish(t2, k1) : K1`

4.6.8 Server login

- *Receive uid and h1*
- `server_auth_start(uid, h1, file): (e, Y, h2)`
- *Response e, Y and h2*
- *Receive t1 and X*
- `server_auth_finish(X, Y, A, y, b, t1, file): (K2, t2)`
- *Response t2*

4.6.9 Structure

Ciphersuite file

4.7 Dependencies/Libraries choices

- OPRF : https://crates.io/crates/voprf
- Curve : https://crates.io/crates/curve25519-dalek (Support Elligator2)

4.8 Code structure

4.9 Interesting functions

4.9.1 Discharge password from RAM directly after use

4.9.2 (Timing attack mitigation)

5 Use case

5.1 Online password manager

Online password manager are among the most sensitive site out there because the compromise of user's data cascade into numerous account compromisation on other services such as email account, social media, banking, etc.

Using an asymmetric PAKE for an online password manager make a lot of sense because the client doesn't have to disclose it's master password to the password manager host. In other words, the client doesn't have to trust the password manager host to not decrypt its personal data and/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 use aPAKE (SRP).

5.2 Other use case

More generally, using an aPAKE makes a lot of sense on application where the server-side stored user's data shouldn't be visible to the server (server doesn't process the data, online backup, online wallet?, secure vault, password manager, etc.). This is archived with encryption and so require an encryption key for the client. Depending on the client, it is not feasable to store an additional symmetric key because it has to be securely stored (see HSM) which cause problem of portability and key recovery. For example, for an online encrypted backup of a laptop or smartphone, if the user loose 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 differents passwords but this is generally avoided because of bad user experience). Using a classical authentication method, the server store the user's encrypted sensible 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 an 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 these case senario. The server NEVER see the user's password so he cannot use it to decrypt user's data.

5.3 Design

client —uid, OPRF—> server

client <—e, Y, OPRF— server

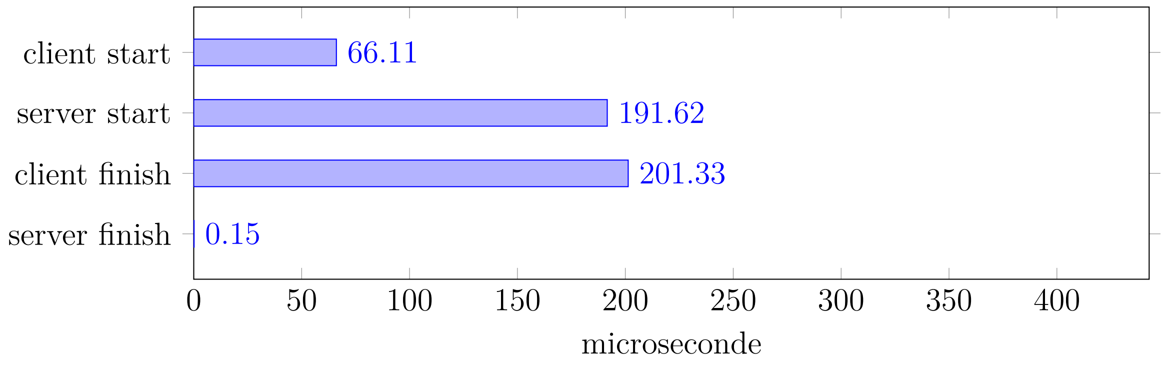
client —X, t1 —> server

client <—t2, data— server

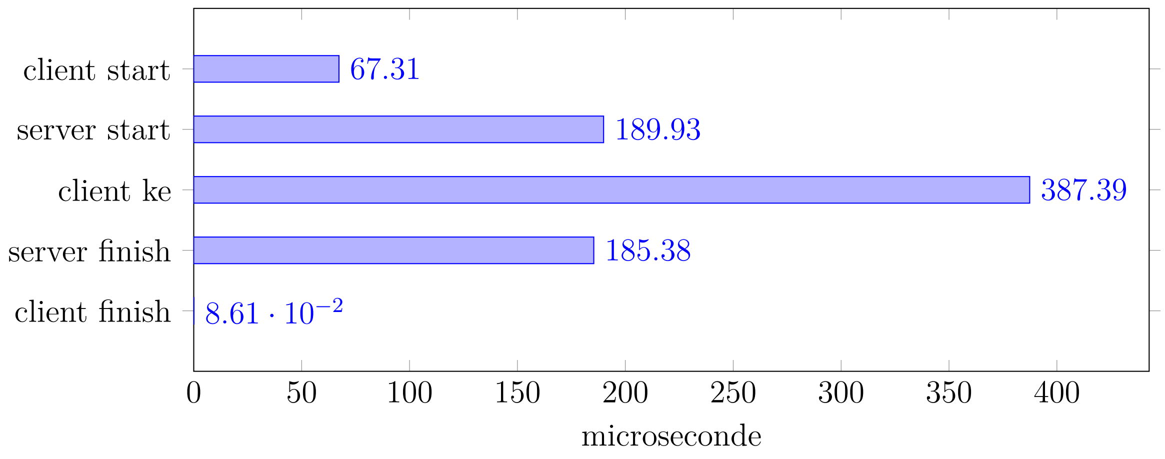
6 Results

6.1 KHAPE endpoints

Registration



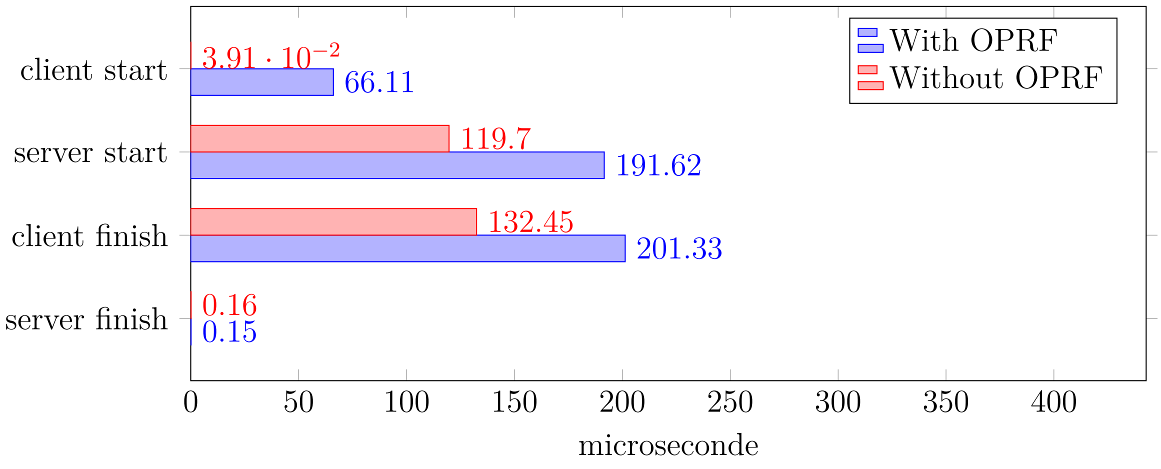
Authentication



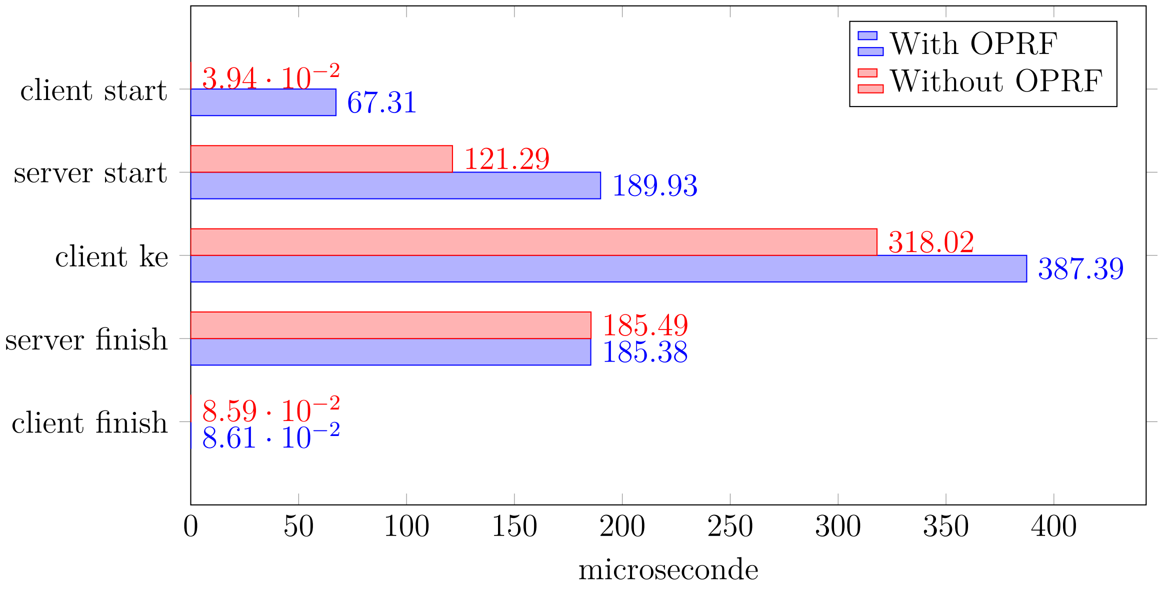
6.1.1 Ideal Cipher vs AES

6.1.2 With OPRF vs without OPRF

Registration

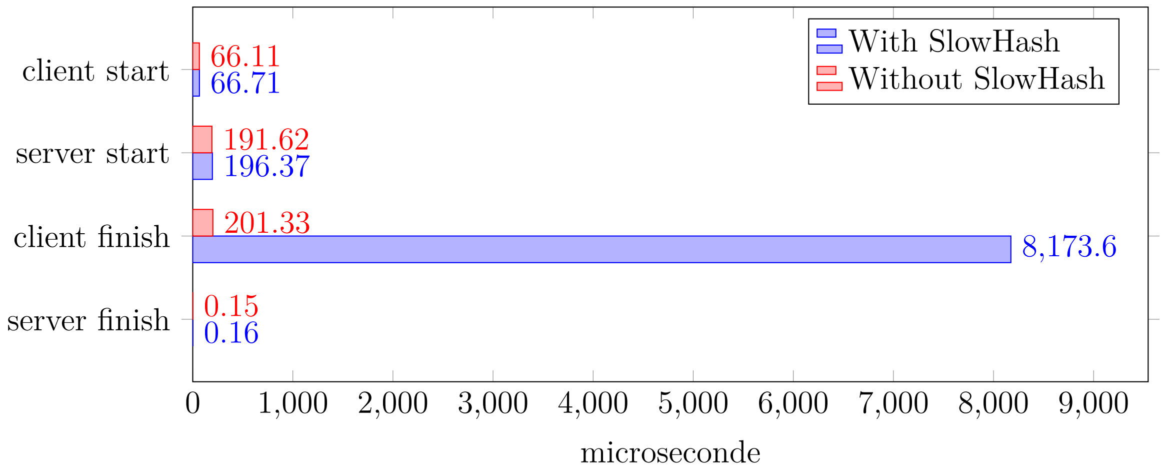


Authentication

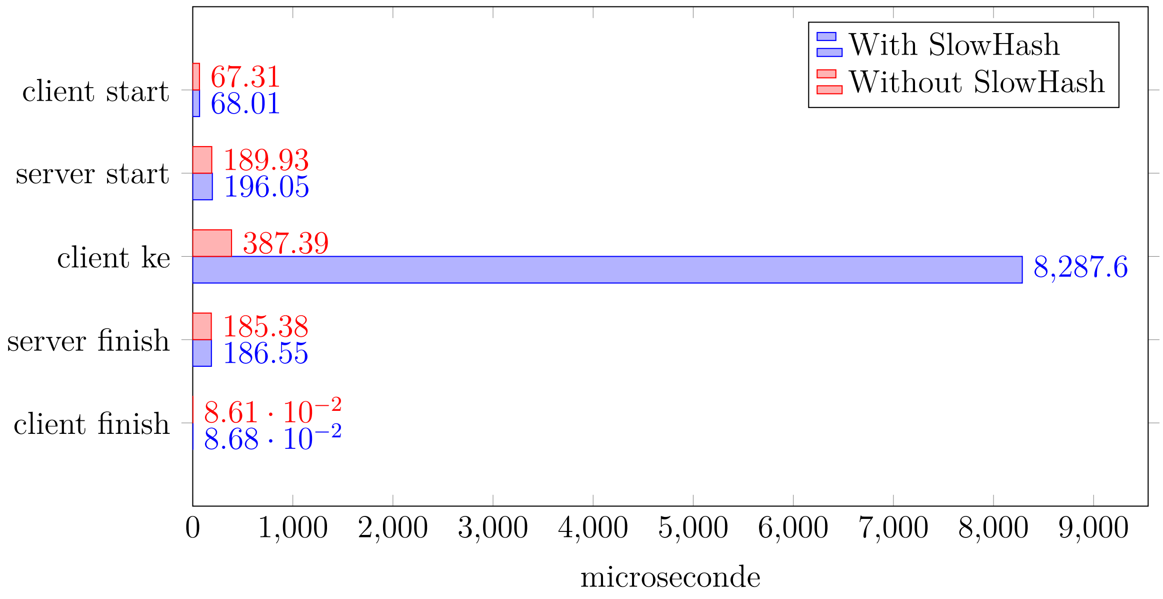


6.1.3 With SlowHash vs without SlowHash

Registration



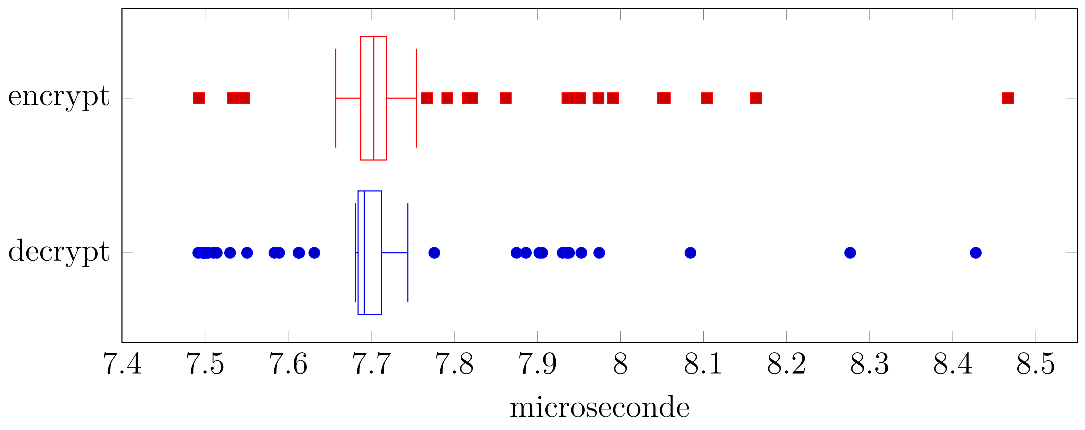
Authentication



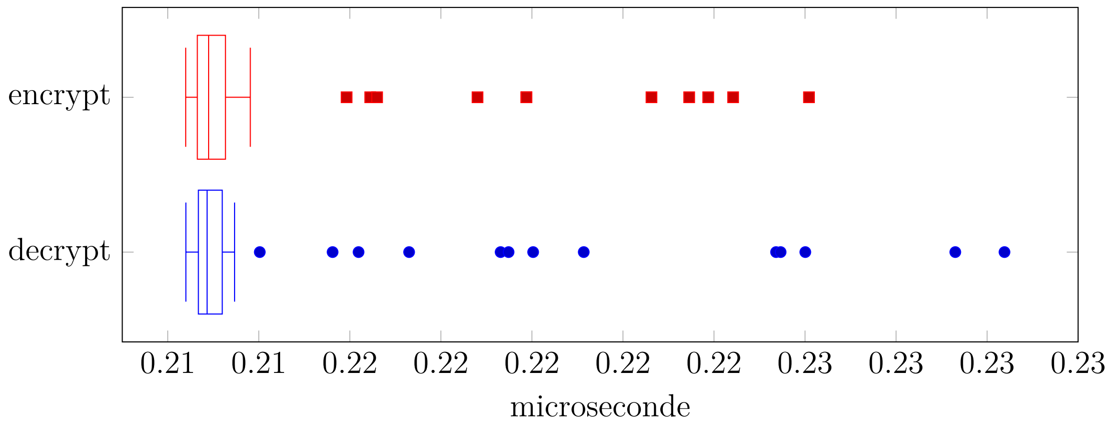
6.2 Components benchmark

6.2.1 Encryption (vs AES)

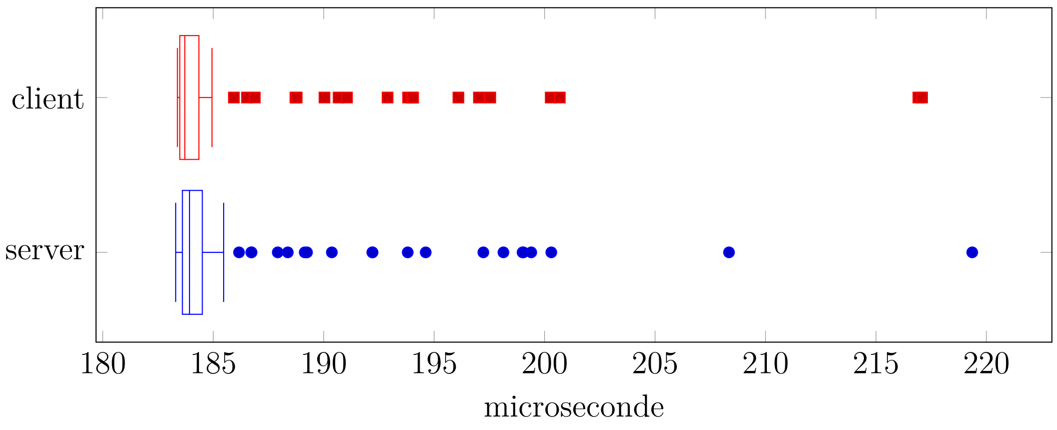
Ideal cipher



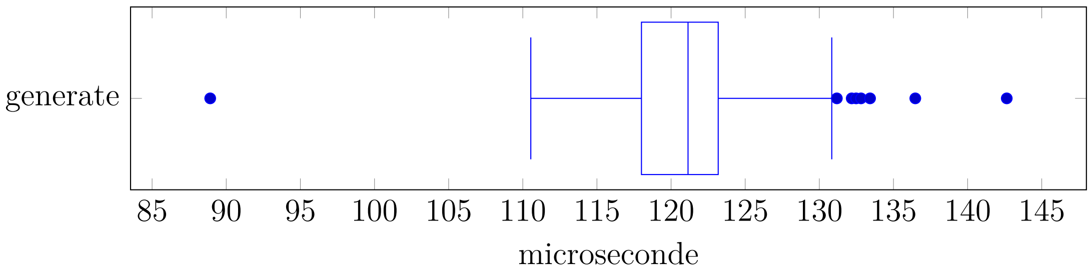
AES256-CTR

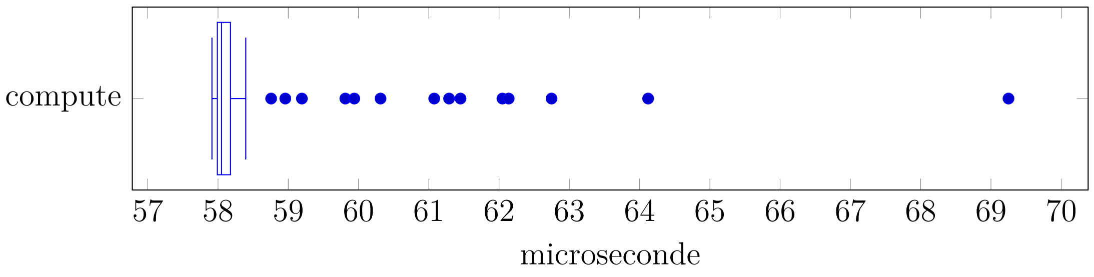


6.2.2 TripleDH



6.2.3 Key generation

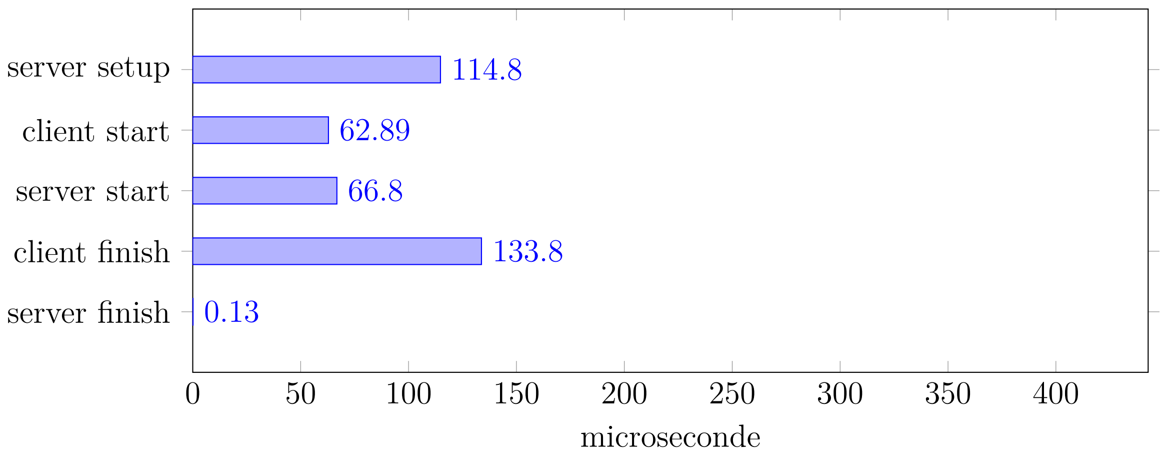




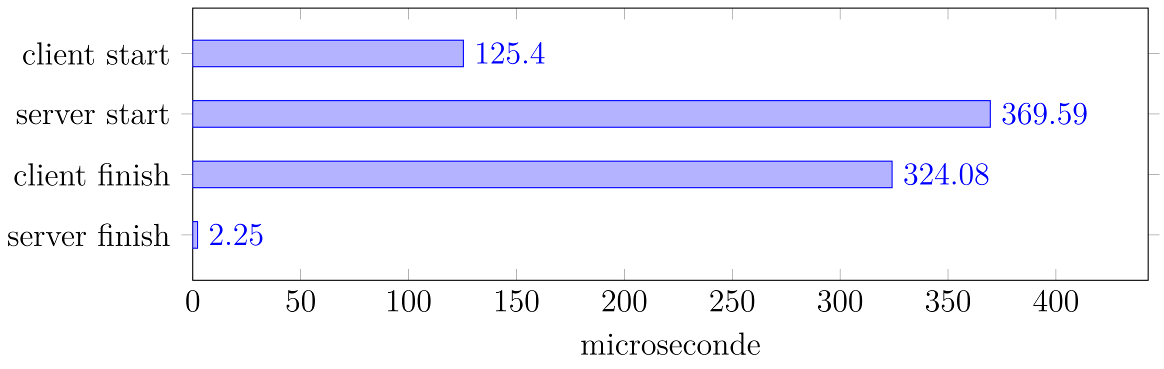
6.3 OPAQUE endpoints

Note that OPAQUE has only 4 functions for the authentication (where KHAPE needs 5) and for the registration, the server compute a server setup where he generate his key pair (in KHAPE, this is done in server start). Benchmark is computed with a ciphersuite that is close to the primitive used for KHAPE : RistrettoPoint for OPRF, MontgomeryPoint for KE, TripleDH, Sha3 for hashing. Note that the default ciphersuite for OPAQUE use Sha2 instead of Sha3 and RistrettoPoint for both OPRF and KE. Performance are relatively similar between the two ciphersuites.

Registration

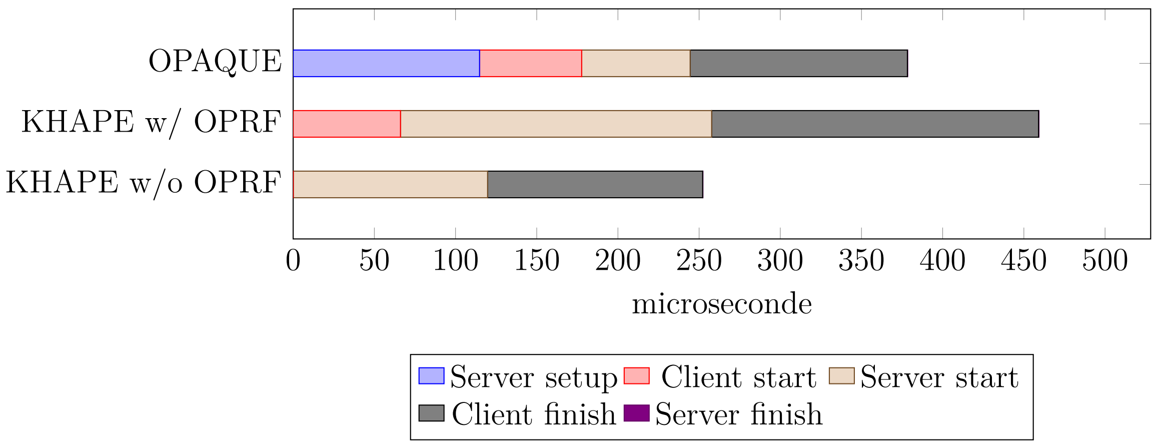


Authentication

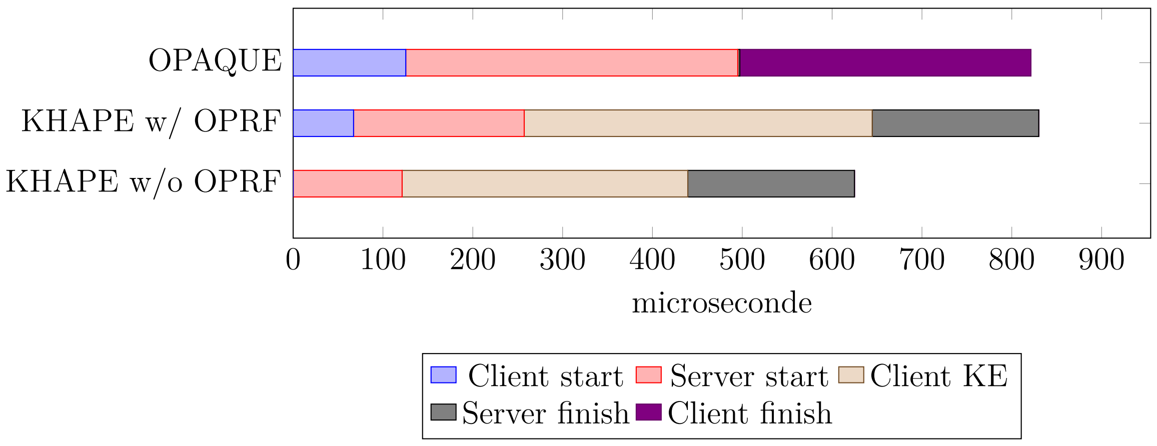


6.4 OPAQUE vs KHAPE

Registration



Authentication



6.5 Message size

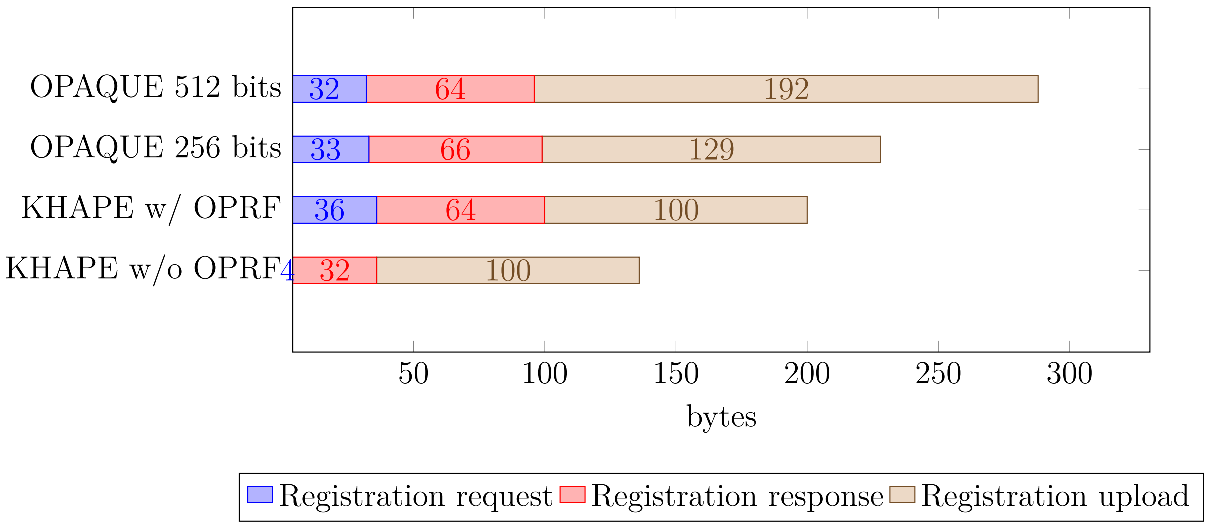
KHAPE not serialized OPAQUE value are taken from the tests vector outputs.

To match the value of OPAQUE's tests vector, a uid of 4 bytes and a password of 25 bytes is inputted

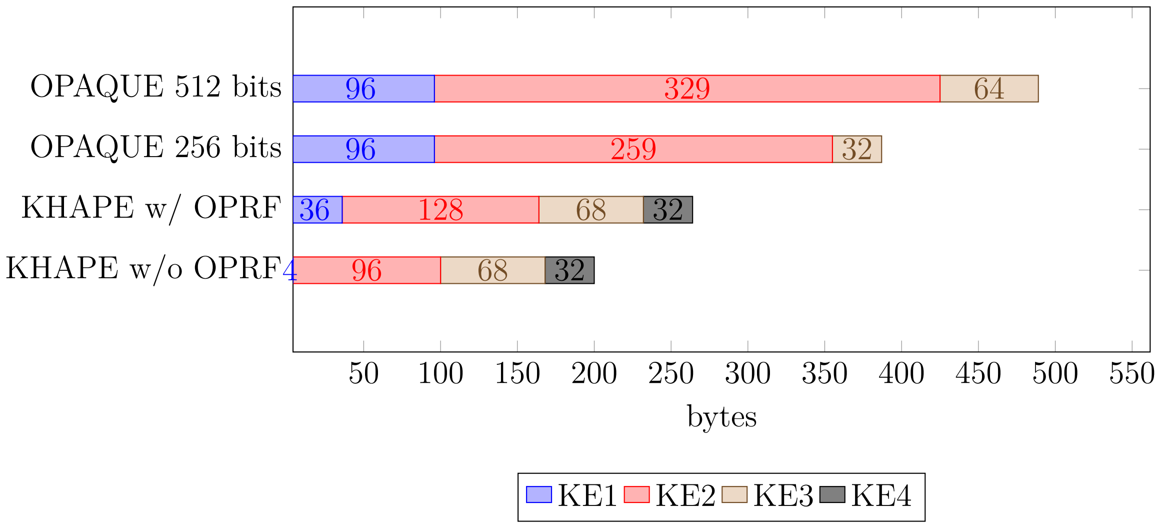
For OPAQUE, the output size depends on the primitive used (ciphersuite) but the big difference is on the choice of the hashing function output size (and so MAC and

KDF). It is either 256 bits or 512 bits. P256_XMD with 256 bits hash, MAC and KDF (!! size of everything is 32 bytes except for the size of public key which is 33 bytes) Ristretto255 with 512 bits hash, MAC and KDF

Registration



Authentication



7 | Conclusion

7.1 Future work

7.2 Ideas not implemented

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Tatiana Bradley. OPAQUE: the best passwords never leave your device, 2020.

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

[10] 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.

[11] Rick Fillion. Developers: How we use srp, and you can too, 2018.

[12] 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.

[13] Matthew Green. Let's talk about PAKE, 2018.

[14] Matthew Green. Should you use SRP ?, 2018.

[15] 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.

[16] 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.

[17] Hugo Krawczyk, D. Bourdrez, K. Lewi, and C.A. Wood. *The OPAQUE Asymmetric PAKE Protocol*. Internet Engineering Task Force, Fremont, California, USA, draft-irtf-cfrg-opaque-07 edition, 2021.

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

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

[20] 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.

[21] 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.

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

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

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

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

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

[27] 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

2.1 Schema notation. 5
2.2 Login process with EKE (DH-EKE) protocol. 7
2.3 Login process with SRP-6a protocol. 9
2.4 Login process with generic OPAQUE (OPRF-AKE) protocol. 11
2.5 Login process with generic KHAPE protocol. 14

List of Tables