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, October 12, 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

1 Introduction

1.1 Problematic

1.1.1 Authentication

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 need a way to prove to the server that he is indeed the person he pretend 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. password, PIN, ...), something that the user has (e.g. digital certificate, OTP token device, smartphone, ...) or something that the user is (e.g. fingerprint, 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 store for the specific user.

This means that the server has to knows and store this sensible value before authentication (generally during register).

Traditionally on websites and softwares, passwords are used as 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, since the server store the passwords, the attacker immediately obtain access to all passwords. This means that he can impersonate every user.

To avoid this scenario, numerous technique has been developed. - memory-hard password hashing function (scrypt, Argon2, etc...) - salt - pepper

These techniques improve the security of storing password but they doesn't address a 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 password 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 to manages password 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” [2]. Now-a-day, majority of password use require that the password is sent in cleartext.

Even if the channel between the client and the server is appropriatly secured, generally with TLS (Can also fail: PKI attack, cert missconfiguration, ... TODO), 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 forget. While the password in 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 (Spectre, 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. (TODO examples: WebAuthn).

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 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.

1.1.2 Password-Authenticated Key Exchange

PAKEs at the rescue Password-Authenticated Key Exchange (PAKE) is a cryptographic primitive. There is 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. In majority of case, the server doesn't want to setup a key exchange with the client, he just want to authenticate the client.

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" [7].

only vulnerable to inevitable attacks (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.

2 | State of the art

2.1 History of PAKEs

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

OKE

SNAPI

PEKEP

2.1.1 Symmetric PAKE

2.1.2 Asymmetric PAKE

SRP

OPAQUE (2018)

KHAPE (2021)

2.2 Main PAKEs

2.2.1 OPAQUE

Design Jarecki and al. [6]. introduce the definition of Strong aPake (SaPAKE): an aPake secure against pre-computation attacks.

They provides two modular constructions, called the OPAQUE protocol that allow to builds SaPake protocols. The first construction allow to enhance any aPake to a SaPAKE while the second allow to enhance any Authenticated Key-Exchange (AKE) protocol (that are secure against KCI attacks) to an 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 allow to secure authentication from the simplest applications to the most sensitive ones.

Construction Figure 2.1 shows the OPAQUE protocol using OPRF and AKE during login process. The steps are the following :

(1) Generate a random value r to blind the hash of password so that the server cannot retrieve the password from the mapping. (2) Send result to server (3) Server add the salt to the password (4) Client calculate the exponant of the inverse of r to unblind the value. He canno't retrieve salt. (5) With the secret salt salt2, client compute secret key sk. (6) Server send encrypted keys ek to clients. ek contains server's public key and client's private key encrypted with sk. (7) If the password entered is correct, client can use sk to decrypt ek and retrieve his private key privU (8) With both keys, clients and server can run an authenticated key exchange for mutual authentication.

Additional features "Supports user-side password hardening" "has a built-in facility for password-based storage-and-retrieval of secrets and credentials" "accommodates a user-transparent server-side threshold implementation" "far more secure alternative to the practice of deriving low-entropy secrets directly from a user's password"

Register The client registration is the only part of the protocol that require a secure channel where both party can authenticated each other.

The protocol is proposed with a server-side registration where the client sends his password through the secure channel. The server generate a salt and compute OPRF function with client's password and salt. Server also generate 2 privates keys (one for the client and one for the server) and their corresponding public key. He encrypt client's private key and server's public key with OPRF output as a key and store the ciphertext.

$C = Encrypt_r w_(client'sprivatekey|server'spublickey)$

This method is not ideal as it require the user to sends it's cleartext password to the server making it vulnerable to miss-handling or server-side vulnerabilities discussed in the introduction.

[6] also note that ideally, one want 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 generate a public/private key pair and the server do the same. Server sends his public key to the client. Client encrypt 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 term of security.

However, this also come 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 enter it's 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 use these keys to run a authenticated key exchange with the server (like HMQV ?).

In the other hand, if the password is wrong, the OPRF output is totally different and the ciphertext decryption make the keys uncorrect and the server will refuse it during the key exchange (?).

2.2.2 KHAPE

Introduction OPAQUE security rely entirely on the strength of the OPRF. If OPRF get 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 is currently no known quantum-safe OPRFs.

KHAPE (for Key-Hiding Asymmetric PakE) [5] is a variant to the OPAQUE protocol. Instead of using OPRF as a main tool to archive security, it become an optional part of the protocol and KHAPE use two other concepts to archive security: non-commiting 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 [6] using OPRF.

So OPRF is optional with KHAPE and just allow to make it a SaPAKE. In addition, it also allow to use OPRF features such as server-side threshold implementation that doesn't require client's change. If OPRF fails, KHAPE just loss these functionalities but the rest of the security remain in contrary to OPAQUE.

In term of implementation, [5] prove that 3DH and HMQV are key-hiding AKE and can be used in KHAPE It also shows that (some KEM-based AKE like) SKEME can be adapted to archive similar result "if instantiated with a key-hiding KEM".

Mains differences with OPAQUE - KHAPE is computationally more performant than OPAQUE when the OPRF is not used and the performances become similar when OPRF is used. - "KHAPE seems more conducive to post-quantum security via post-quantum key-hiding KEMs."

- KHAPE allow less AKEs (no signature-based protocols such as SIGMA) - KHAPE use “idea cipher model”, OPAQUE use “random oracle model” (stronger ?)

- with OPAQUE the client learns first if an authentication attempt is succeed or not. with KHAPE, the server learns first which make it easier to count failed attempt in case of an online password guessing attack. - KHAPE (without OPRF) cost is $=$ same cost as KE cost where OPAQUE cost is about the double of KE cost

Construction Figure 2.2 shows the KHAPE protocol during login process. The steps are the following :

Both client and server needs a public/private key pair to compute a AKE. The client encrypt it's credentials (his private key and server's public key) and store it in the server.

(1) Optionally, an OPRF can be used to archive Strong aPAKE following the aPAKE to SaPAKE compiler using OPRF from [6]. The OPRF take the client's password and server's salt as an input. Client use the output in place of his password for the rest of the protocol. (2) Server send the client's encrypted envelop containing the client's private key and server's public key. (3) Client decrypt the ciphertext using Ideal Cipher encryption schema. He use his password or OPRF output as a key. (4) Both party 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 want to login, the server sends its encrypted credentials and the client use it's password to decrypt the credentials (with or without OPRF depending on the implementation). Then he can us 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 2.2.1. The protocol propose a server-side register which is less than ideal because the server can see client's password and client's private key in cleartext at register.

Instead, the paper propose a client-side registration process.

2.3 Comparing mains solutions

This section compare the mains PAKEs on their security guarantees and performances. Details and comments on each criteria can be found on Section 2.3.1.

#	**Criteria**	**EKE**	**SRP**	**OPAQUE**	**KHAPE**
10	Server doesn't store password in cleartext	x	x	Yes	Yes
1	Avoid sending cleartext password to the server	x	x	Yes (register?)	Yes (register?)
2	Secure against pre-computation attacks	x	x	Yes	Yes, if using OPRF
3	Server doesn't send salt in cleartext	x	x	Yes	– (no salt)
4	Forward secrecy	x	x	Yes, Full FS	Yes, Full FS
5	Mutual authentication	x	x	Yes	Yes ?
6	PKI-free	x	x	Yes, except during register	Yes, except during register
7	User-side password hardening	x	x	Yes	x
8	Built-in mechanism to store client's secrets on the server	x	x	Yes	x
9	Server threshold implementation	x	x	Yes, user-transparent	Yes, if using OPRF
11	Resistant upon Oblivious PRF compromise	x	x	No, entire security is compromised	Fall back to non-strong aPAKE
12	Internet standard	x	x	Draft	x
13	Security proof	x	x	Yes, in a very strong model (random oracle model ?)	Yes (idea cipher model)
14	Easily adaptable to elliptic curves	x	x	Yes	x
15	Number of messages	x	x	3	4 (3 if client initiate)
16	Number of exponentiations	x	x	3 or 4 ?	x
17	Patented	Yes, expired in 2011 ?	x	No	No
18	Year published	x	x	2018	2021
19	Got broken	x	x	x	x

2.3.1 Details

10. Server doesn't store password in cleartext. This is the main security property of asymmetric PAKE [3]. Server doesn't have to store password 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.

1. Avoid sending password in cleartext to the server. Even though it seems similar to criteria 1, it's not. Criteria 1 is about password storage but this criteria is about password transmission. Transmissions and storage of the password are vulnerable to different attacks vectors. The server don't receive password in cleartext which avoid any miss-handling vulnerabilities such as logging or caching cleartext password on the server.

2. Secure against pre-computation attacks. This is the main security property of Strong aPAKE [6]. The server doesn't leak any data (generally the salt) that could allow an attacker to perform a pre-computation attack. This attack allow an attacker to compute a table *before* the server even get compromised. Once the attacker succeed in compromising the server, he can use the pre-computed table to retrieve the passwords *instantaneously*. So this protection force the attacker to perform an offline dictionary attack after successful server compromise.

3. Server doesn't send salt in cleartext. Same as 2.

4. Forward secrecy. In key-exchange protocol, Forward Secrecy (also called Full Forward Secrecy or Perfect Forward Secrecy) ensure that upon compromise of any long term key used to negotiate sessions key, an attacker cannot compromise previous session keys. In details 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.

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

6. PKI-free. The transmissions between client and server doesn't require to be secured with 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. User can use password hardening technique to increase the cost of an offline attacks if the server get compromised. This is done by using ressource-heavy functions such as Scrypt [8] or Argon2 [1] instead of computing a simple and efficient hash. These functions allows to drastically slows down hashing process and so making offline attacks and online guessing attack much slower.

8. Built-in mechanism to store client's secrets on the server. OPAQUE "has a built-in facility for password-based storage-and-retrieval of secrets and credentials"

9. Server threshold implementation. OPAQUE "accommodates a user-transparent server-side threshold implementation"

11. Resistant upon Oblivious PRF compromise. If OPRF breaks for exemple 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” [5]. OPAQUE use OPRF as a main tool to builds Strong aPAKE. If OPRF breaks, the client's password is vulnerable to a 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 make KHAPE more resistant to OPRF compromise than OPAQUE.

12. Internet standard.

13. Security proof. OPAQUE "security proof in a very strong model"

14. Easily adaptable to elliptic curves.

15. Number of messages.

16. Number of exponentiations.

17. Patented.

18. Release date.

19. Version.

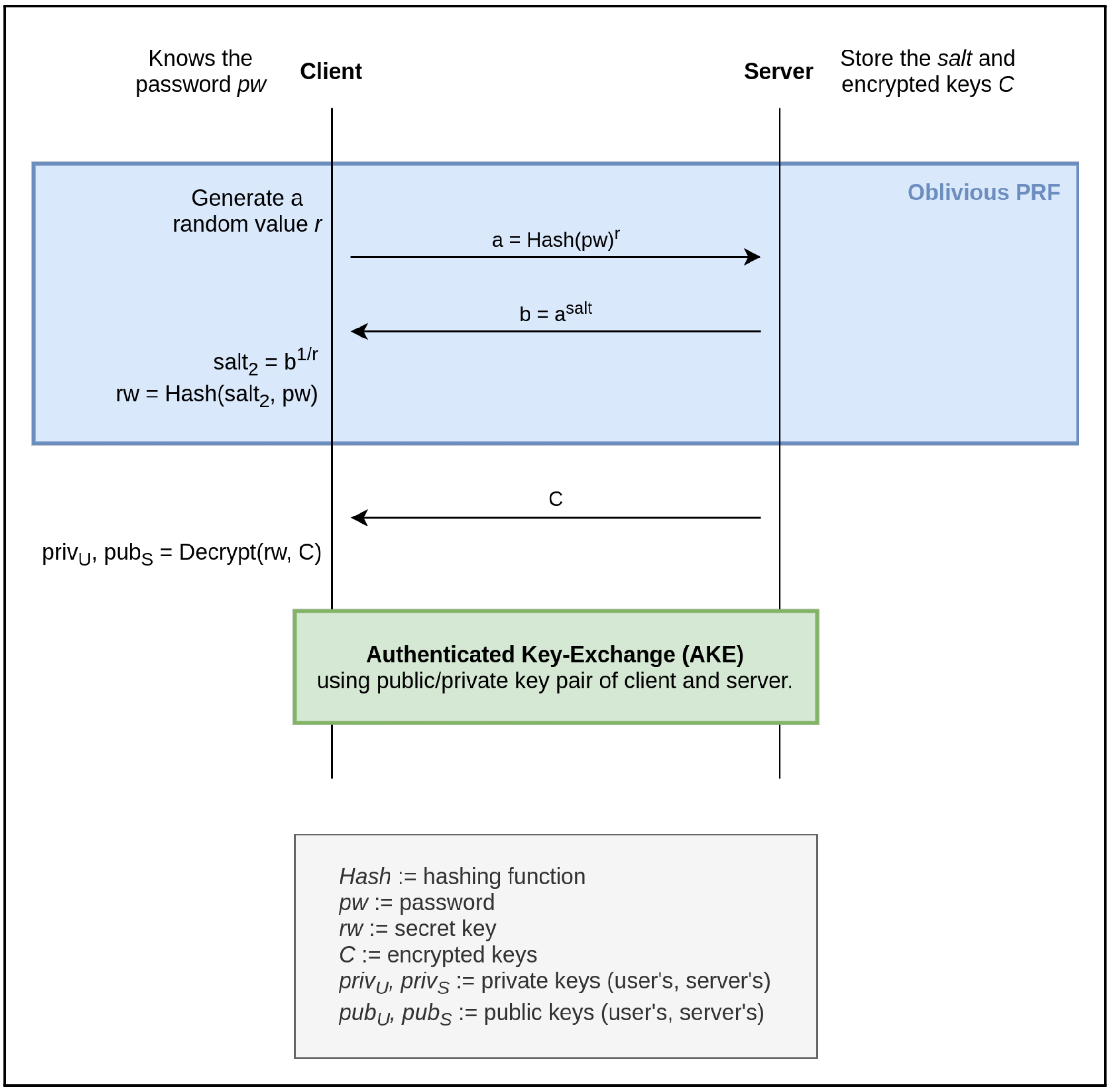
Knows the password *pw*
Client
Server
Store the *salt* and encrypted keys *C*
Oblivious PRF
Generate a random value *r*
$a = Hash(pw)^r$
$b = a^{salt}$
$salt_2 = b^{1/r}$
$rw = Hash(salt_2, pw)$
C
$priv_U, pub_S = Decrypt(rw, C)$
Authenticated Key-Exchange (AKE) using public/private key pair of client and server.
Hash := hashing function
pw := password
rw := secret key
C := encrypted keys
$priv_U, priv_S$:= private keys (user's, server's)
pub_U, pub_S := public keys (user's, server's)

Figure 2.1: Login process with generic OPAQUE (OPRF-AKE) protocol.

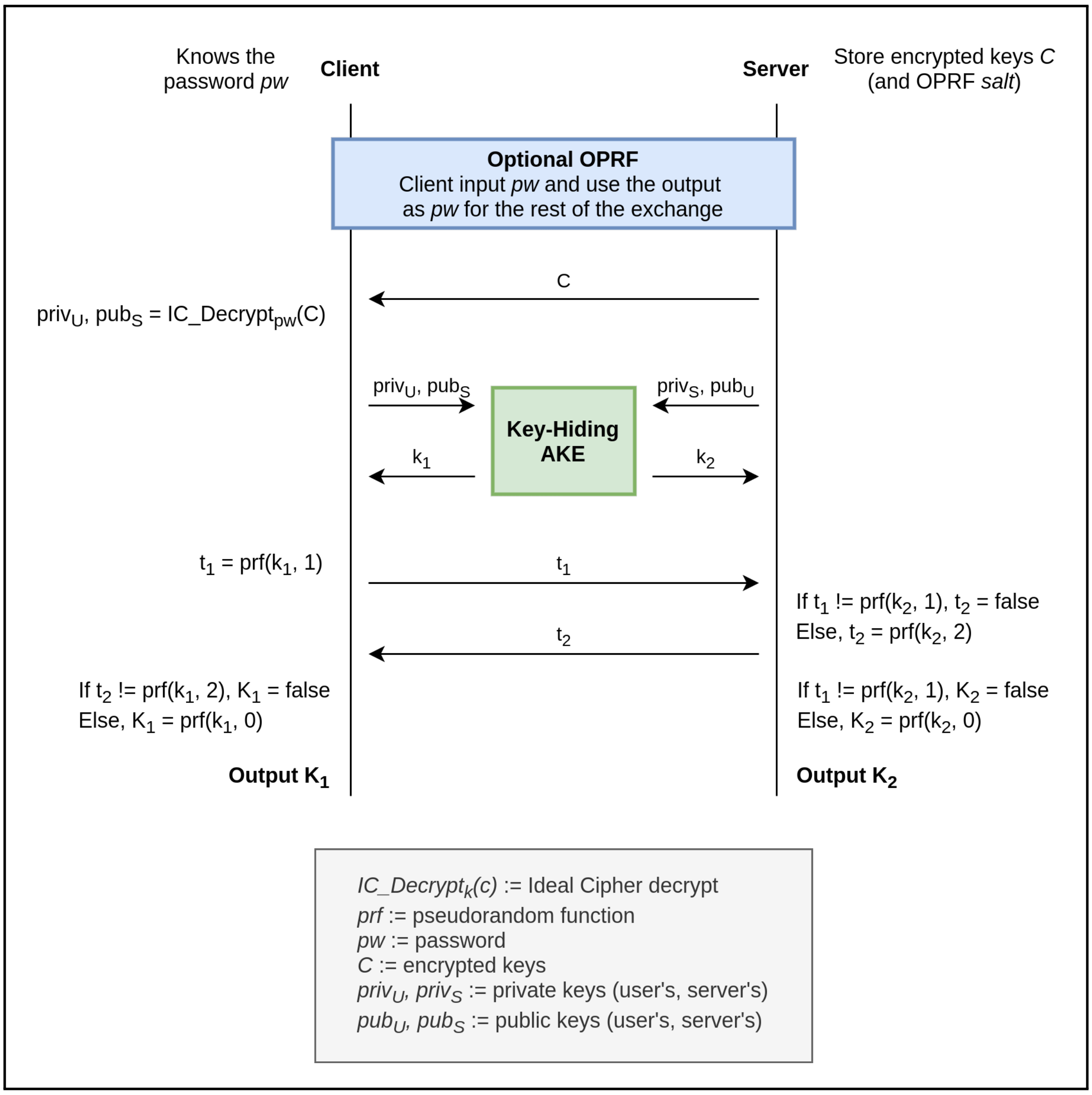
Knows the password *pw*
Client
Server
Store encrypted keys *C* (and OPRF *salt*)
Optional OPRF
Client input *pw* and use the output as *pw* for the rest of the exchange
C
$priv_U, pub_S = IC\_Decrypt_{pw}(C)$
$priv_U, pub_S$
$priv_S, pub_U$
Key-Hiding AKE
k_1
k_2
$t_1 = prf(k_1, 1)$
t_1
t_2
If t_1 != $prf(k_2, 1)$, 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
$IC\_Decrypt_k(c)$:= Ideal Cipher decrypt
prf := pseudorandom function
pw := password
C := encrypted keys
$priv_U$, $priv_S$:= private keys (user's, server's)
pub_U, pub_S := public keys (user's, server's)

Figure 2.2: Login process withn generic KHAPE protocol.

3 | OPAQUE (or) KHAPE

4 | Use case: ...

5 | Implementation

6 | Conclusion

Bibliography

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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-06 edition, 2021.

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

List of Figures

List of Tables