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, October 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

1.1.1 Authentication . . . 1

1.1.2 Password-Authenticated Key Exchange . . . 3

**2 State of the art** **5**

2.1 Notation . . . 5

2.2 History of PAKEs . . . 5

2.2.1 Symmetric PAKE . . . 5

2.2.2 Asymmetric PAKE . . . 5

2.3 Main PAKEs . . . 6

2.3.1 EKE . . . 6

2.3.2 SRP . . . 8

2.3.3 OPAQUE . . . 8

2.3.4 KHAPE . . . 10

2.4 Comparing mains solutions . . . 12

2.4.1 Details . . . 14

**3 OPAQUE (or) KHAPE** **17**

**4 Use case: ...** **19**

**5 Implementation** **21**

**6 Conclusion** **23**

**Bibliography** **25**

# 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 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 technique has been developed. Mainly, adding salt, adding pepper — a secret salt — and using memory-hard password hashing function such as Scrypt [9] or Argon2 [2].

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" [3]. 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 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.

### 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. 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" [8].

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.

# 2 | State of the art

## 2.1 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

### 2.3.1 EKE

**Introduction** EKE (for Encrypted Key Exchange) was proposed in 1992 by Bellovin and Merritt [1] 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. However, this protocol 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 [12].

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

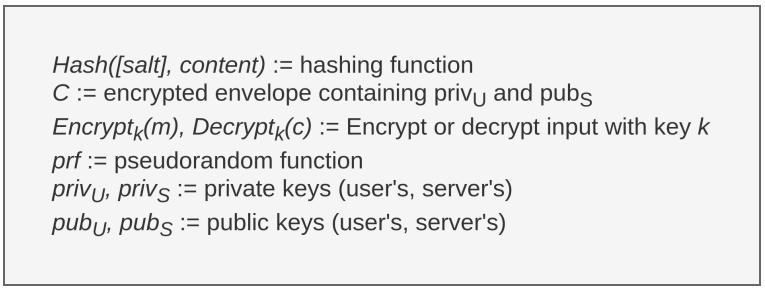


Figure 2.1: Schema notation.

**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.

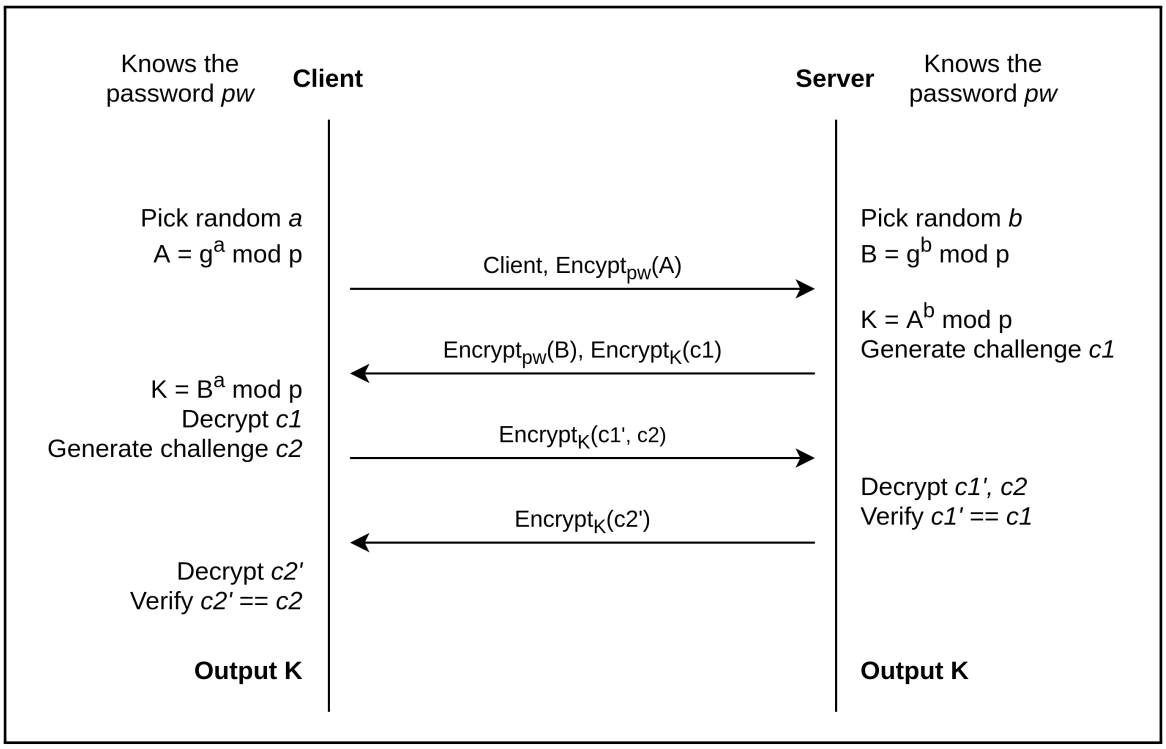


Figure 2.2: Login process with EKE (EKE-DH) protocol. (See Fig. 2.1 for notations)

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

SRP (for Secure Remote Password), 1998.

### 2.3.3 OPAQUE

**Design** Jarecki and al. [7]. 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.

**Construction** The figure 2.3 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.

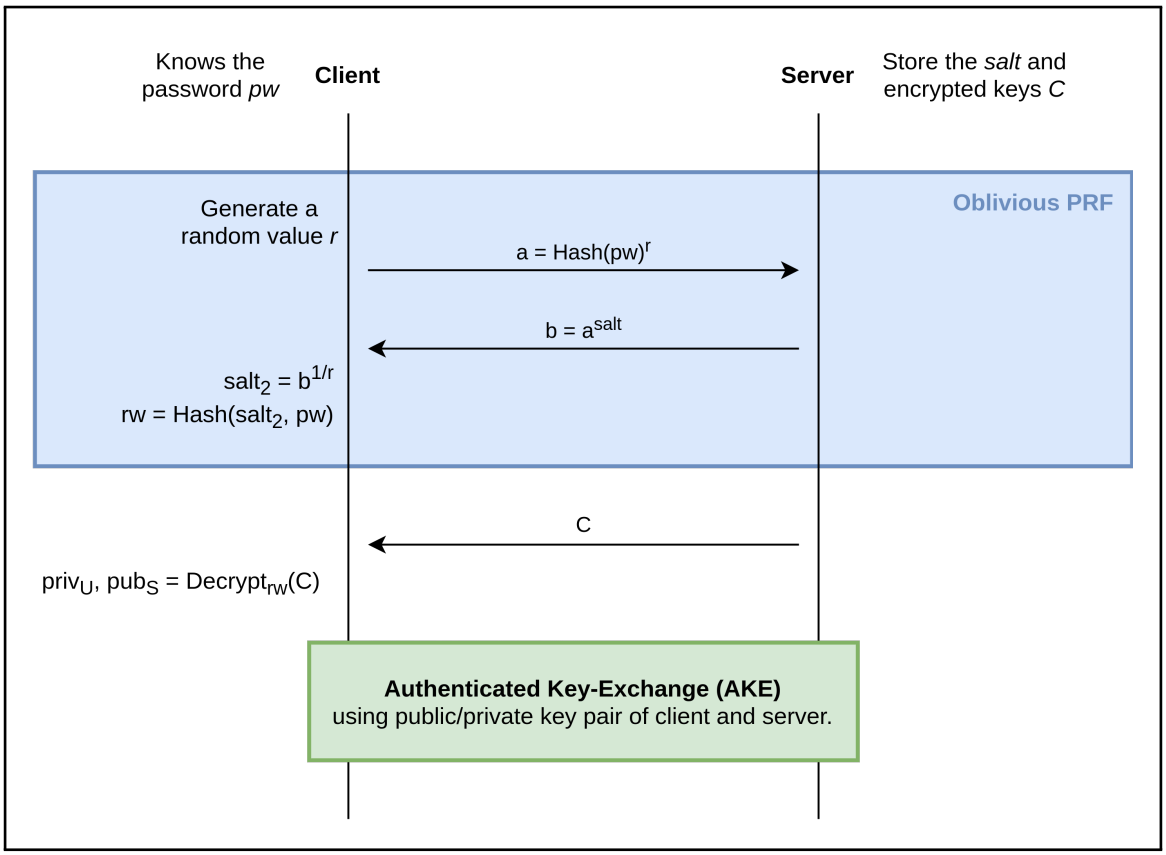


Figure 2.3: Login process with generic OPAQUE (OPRF-AKE) protocol. (See Fig. 2.1 for notations)

**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.

[7] 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) [6] 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 [7] 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.

In terms of implementation, [6] 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 they are instantiated with a key-hiding KEM.

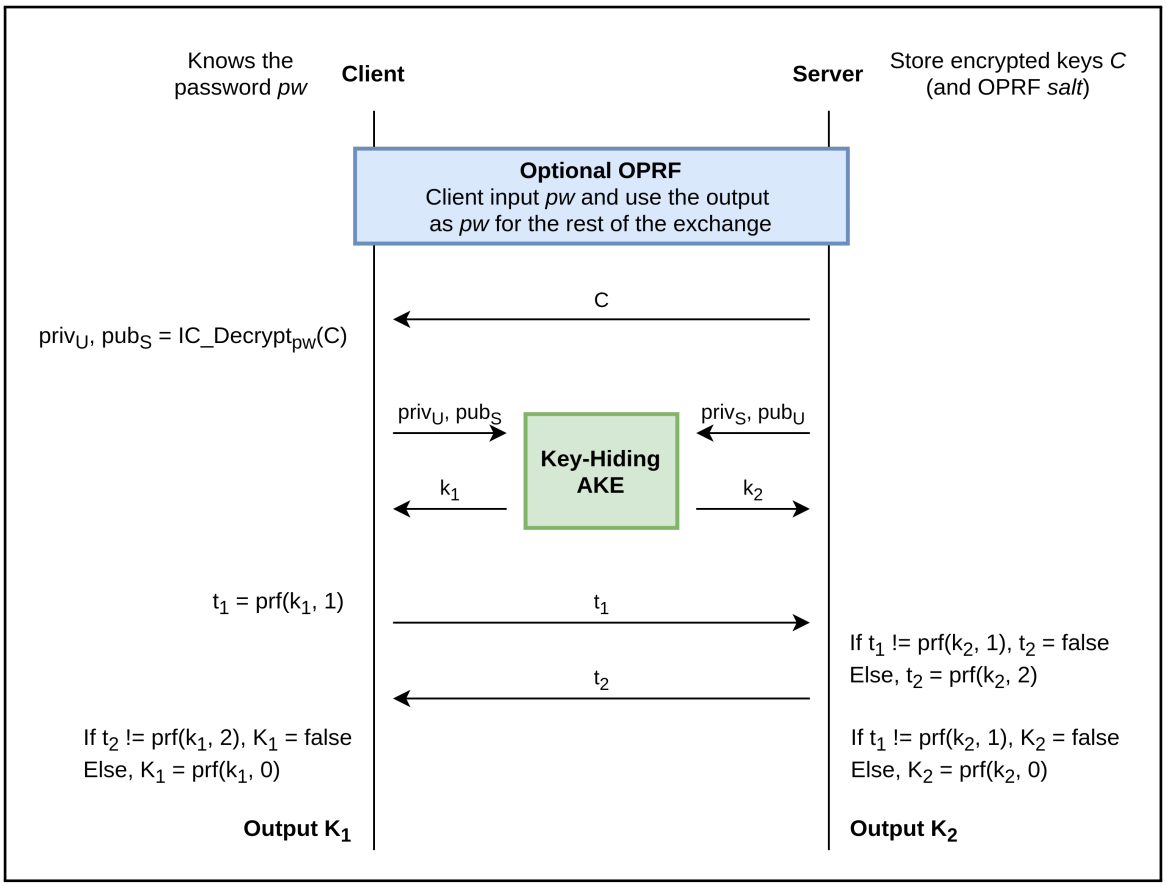


Figure 2.4: Login process with generic KHAPE protocol. (See Fig. 2.1 for notations)

**Construction** Figure 2.4 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 [7]. 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 mains 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.

| # | **Criteria** | **EKE** | **SRP** | **OPAQUE** | **KHAPE** |
|---|---|---|---|---|---|
| 1 | Server doesn't store passwords in cleartext | No | x | Yes | Yes |
| 2 | Avoid sending cleartext password to the server | No | x | Yes[1] | Yes[2] |
| 3 | Secure against pre-computation attacks | - (no hash) | x | Yes | Yes, if using OPRF |
| 4 | Forward secrecy | Yes ? | x | Yes, Full FS | Yes, Full FS |
| 5 | Mutual authentication | Yes | x | Yes | Yes |
| 6 | PKI-free | Yes, except during register | x | Yes, except during register | Yes, except during register |
| 7 | User-side password hardening | No[3] | x | Yes | Yes, if using OPRF |
| 8 | Built-in mechanism to store client's secrets on the server | No | x | Yes | Yes |
| 9 | Server threshold implementation | No | x | Yes, user-transparent | Yes, if using OPRF |
| 11 | Resistant upon Oblivious PRF compromise | - (no OPRF) | x | No, entire security is compromised | Fall back to non-strong aPAKE |
| 12 | Standardization status | RFC for EAP-EKE [10] | x | Internet standard draft [8] | Crypto 2021 Paper [6] |
| 13 | Security proof | No [4] | x | Yes, in a very strong model (random oracle model ?) | Yes (idea cipher model) |

| # | Criteria | EKE | SRP | OPAQUE | KHAPE |
|---|---|---|---|---|---|
| 14 | Easily adaptable to elliptic curves | Yes ? | x | Yes | Yes ? |
| 15 | Number of messages | 4 ? | x | 3 | 4 (3 if client initiate) |
| 16 | Number of exponentiations | 4 ? | x | 3 or 4 ? | 2 + 1 hash-to-curve |
| 20 | Computational cost compared to a KE (see [6] presentation) | 1x | x | 2x | 1x without OPRF, 2x with OPRF |
| 17 | Patented | Yes, expired in 2011 | x | No | No |
| 18 | Year published | 1992 | 1998 | 2018 | 2021 |
| 19 | Got broken | Yes (source ?) | x | x | x |

### 2.4.1 Details

**1. Server doesn't store password in cleartext.** This is the main security property of asymmetric PAKE [4]. 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 password in cleartext to the server.** Even though it seems similar to criteria 1, it's not. Criterion 1 is about password storage, but this criterion is about password transmission. Transmissions and storage of the password 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.

**3. Secure against pre-computation attacks.** This is the main security property of Strong aPAKE [7]. The server doesn't leak any data (generally the salt) that could allow an attacker to perform a pre-computation attack. 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.

**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 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 [9] or Argon2 [2] 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.

**8. Built-in mechanism to store client's secrets on the server.**

**9. Server threshold implementation.**

**11. Resistant upon Oblivious PRF compromise.** 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" [6]. 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.

**12. Standardization status.**

**13. Security proof.**

**14. Easily adaptable to elliptic curves.**

**15. Number of messages.**

**16. Number of exponentiations.**

**17. Patented.**

**18. Release date.**

**19. Version.**

# 3 | OPAQUE (or) KHAPE

# 4 | Use case: ...

# 5 | Implementation

# 6 | Conclusion

# Bibliography

[1] 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.

[2] 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.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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. . . . . 6
2.2 Login process with EKE (EKE-DH) protocol. (See Fig. 2.1 for notations) 7
2.3 Login process with generic OPAQUE (OPRF-AKE) protocol. (See Fig. 2.1 for notations) . . . . . 9
2.4 Login process with generic KHAPE protocol. (See Fig. 2.1 for notations) 11

# List of Tables