Cryptography Engineering

- Lecture 9 (Jan 16, 2024)
- Today's notes:
  - Recall previous contents
  - The OPAQUE protocol
  - Summary on password-based authentication
  - Notes on the final project

- Coding tasks/Homework:
  - Implement the OPAQUE protocol
  - **Bonus:** Implement OPAQUE using sockets

Previous lecture contents

- Welcome back from the Christmas holidays!

- L1: Recall some cryptographic primitives
- L2: Signature and Certificate
- L3: DHKE + Signature & Certificate = TLS handshake
- L4: Secure Messaging, E2EE, X3DH
- L5 & L6: Key chain, Double ratchet = Symmetric ratchet + DH ratchet
- L7: Passwords, Off/Online attacks, TLS + passwords, Salting
- L8: SCRAM (hashed+salted+iterated), Password-based AKE (EKE, SRP)

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The SCRAM protocol
- The EKE protocol
- The SRP protocol

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The SCRAM protocol
- The EKE protocol
- The SRP protocol

- Goal: Authentication via passwords; Resistance to offline attacks.

Previous Password-based Protocols

- **TLS + hashed & salted passwords**
  - Store $(\boldsymbol{r}, \boldsymbol{H}(\boldsymbol{pw}, \boldsymbol{r}))$ in the server, where $\boldsymbol{r}$ is the salt.
  - Transport $\boldsymbol{r}$ to the client, then the client prove its identity by responding $\boldsymbol{H}(\boldsymbol{pw}, \boldsymbol{r})$
  - Encrypted by TLS
- The SCRAM protocol
- The SRP protocol

Previous Password-based Protocols

- TLS + hashed & salted passwords
- **The TLS + SCRAM protocol**
  - Store $\left(\boldsymbol{r}, \boldsymbol{n}, \boldsymbol{H}^{\boldsymbol{n}}(\boldsymbol{pw}, \boldsymbol{r})\right)$ in the server, where $\boldsymbol{r}$ is the salt and $\boldsymbol{n}$ is the number of iterations.
  - Transport $\boldsymbol{r}$ and $\boldsymbol{n}$ to the client, then the client prove its identity by responding $\boldsymbol{H}^{\boldsymbol{n}}(\boldsymbol{pw}, \boldsymbol{r})$
  - Encrypted by TLS
- The SRP protocol

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The TLS + SCRAM protocol
- **The SRP protocol**
  - Store $(\boldsymbol{r}, \boldsymbol{H}(\boldsymbol{pw}, \boldsymbol{r}))$ in the server, where $\boldsymbol{r}$ is the salt.
  - Password-based AKE:
    - Security guarantee even if the certificate is fake or the TLS connection is insecure.
  - Enhanced security via integrating with TLS

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The TLS + SCRAM protocol
- The SRP protocol

- Advantage of storing hashed-salted passwords:
    1. Avoid cross-system leakage

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The TLS + SCRAM protocol
- The SRP protocol

- Advantage of storing hashed-salted passwords:
  1. Avoid cross-system leakage
  2. Increase the time **required to recover the password after leakage**.

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The TLS + SCRAM protocol
- The SRP protocol

Storage	Required Time after leakage
Plain pw	$\mathbf{O(1)}$
H(pw)	$\mathbf{O(\|D\|)}$
r, H(pw, r)	$\mathbf{O(\|D\|)}$

- Advantage of storing hashed-salted passwords:
  1. Avoid cross-system leakage
  2. Increase the time **required to recover the password after leakage**.

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The TLS + SCRAM protocol
- The SRP protocol

Storage	Required Time after leakage
Plain pw	$\mathbf{O(1)}$
H(pw)	$\mathbf{O(\|D\|)}$
r, H(pw, r)	$\mathbf{O(\|D\|)}$

- Advantage of storing hashed-salted passwords:
  1. Avoid cross-system leakage
  2. Increase the time **required to recover the password after leakage**.

This is also important in practice, e.g., notifying users to change their passwords after the leakage.

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The TLS + SCRAM protocol
- The SRP protocol

Storage	Required Time after leakage
Plain pw	$\mathbf{O}(\mathbf{1})$
H(pw)	$\mathbf{O}(\lvert \boldsymbol{D} \rvert)$
r, H(pw, r)	$\mathbf{O}(\lvert \boldsymbol{D} \rvert)$

- Advantage of storing hashed-salted passwords:
  1. Avoid cross-system leakage
  2. Increase the time **required to recover the password after leakage**.

- All protocols **reveal salt** (and the number of iterations) during the execution…

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The TLS + SCRAM protocol
- The SRP protocol

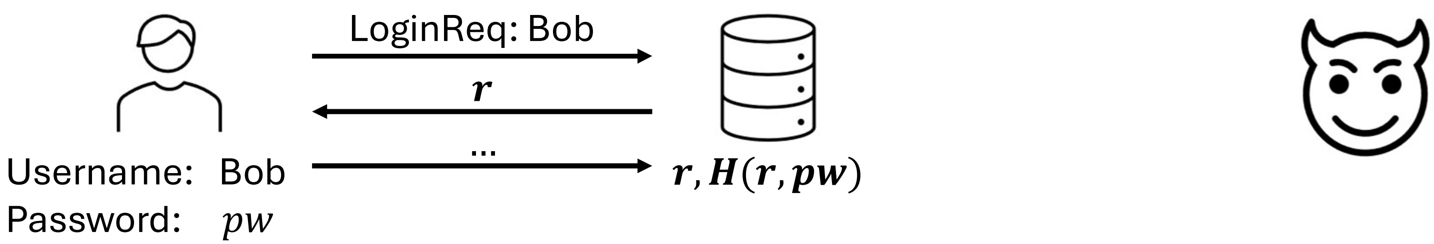
Storage	Required Time after leakage
Plain pw	$\mathbf{O(1)}$
H(pw)	$\mathbf{O}(\lvert\boldsymbol{D}\rvert)$
r, H(pw, r)	$\mathbf{O}(\lvert\boldsymbol{D}\rvert)$

- Advantage of storing hashed-salted passwords:
  1. Avoid cross-system leakage
  2. Increase the time **required to recover the password after leakage**.

- All protocols **reveal salt** (and the number of iterations) during the execution...
  - May lead to **Precomputation Attacks**
  - $\mathbf{O}(\lvert\boldsymbol{D}\rvert) \rightarrow \mathbf{O}(\log\lvert\boldsymbol{D}\rvert)$ or even $\mathbf{O}(1)$

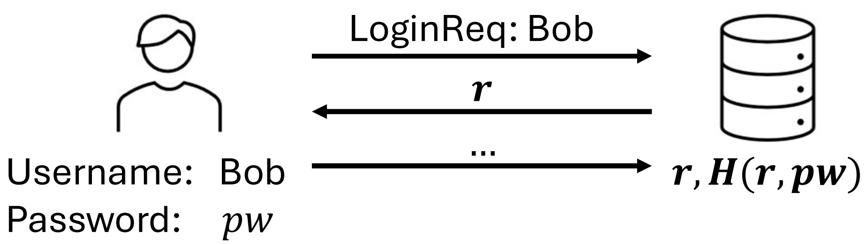
Precomputation Attacks on Passwords

- Suppose that the password is stored by hashing and salting.
  - **The adversary can learn the salt in some easy ways…**



Precomputation Attacks on Passwords

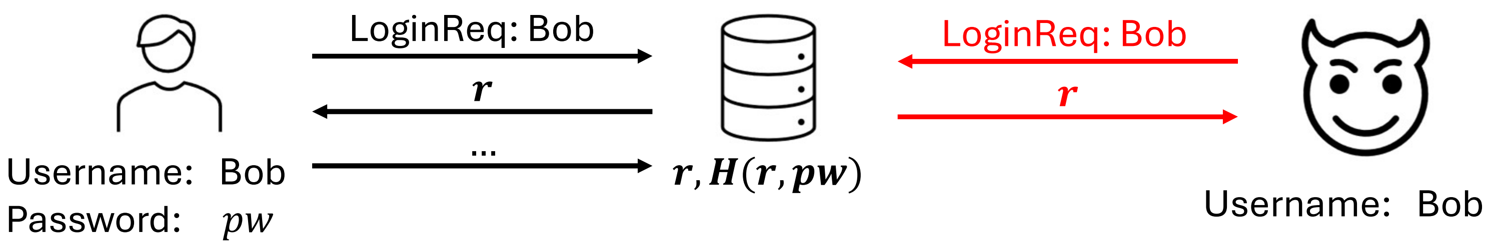
- Suppose that the password is stored by hashing and salting.
  - **The adversary can learn the salt in some easy ways…**



Suppose that the adversary knows the username…

Precomputation Attacks on Passwords

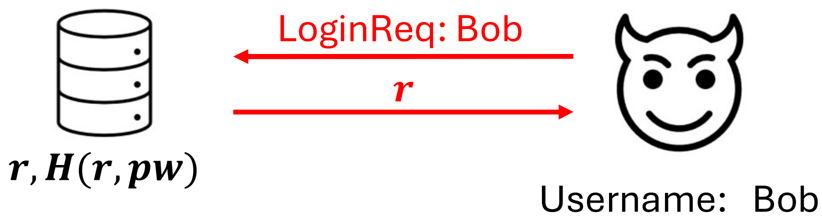
- Suppose that the password is stored by hashing and salting.
  - **The adversary can learn the salt in some easy ways...**



Suppose that the adversary knows the username...
Then it can get the salt...

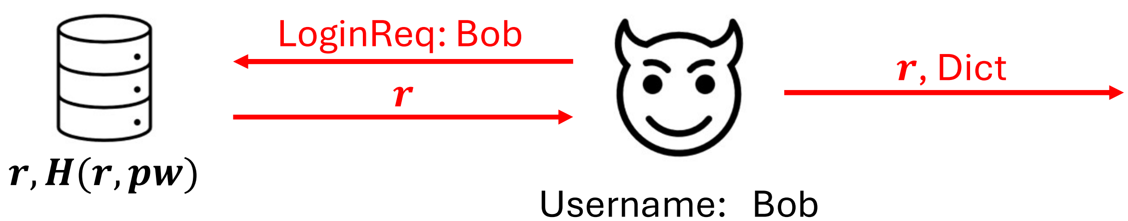
Precomputation Attacks on Passwords

- Suppose that the password is stored by hashing and salting.
  - The adversary can learn the salt in some easy ways…
  - **Precompute a table containing all hashed passwords with the same salt**:



Precomputation Attacks on Passwords

- Suppose that the password is stored by hashing and salting.
  - The adversary can learn the salt in some easy ways…
  - **Precompute a table containing all hashed passwords with the same salt**:

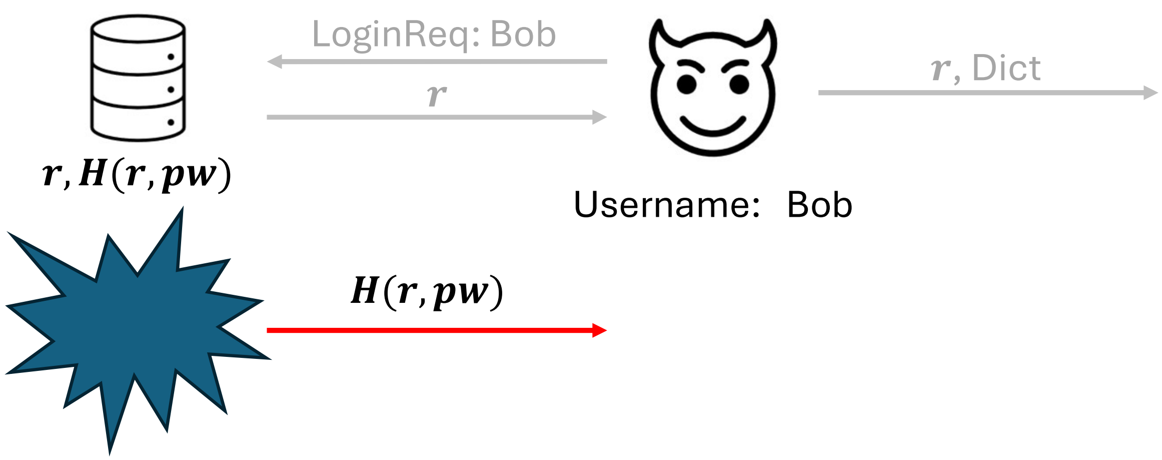


$pw \in$ Dict	**The H(pw, r) values**
pw_1	$\mathbf{H}(pw_1, r)$
pw_2	$\mathbf{H}(pw_2, r)$
pw_3	$\mathbf{H}(pw_3, r)$
pw_4	$\mathbf{H}(pw_4, r)$
…	…

The table can be computed locally…

Precomputation Attacks on Passwords

- Suppose that the password is stored by hashing and salting.
  - The adversary can learn the salt in some easy ways…
  - **Precompute a table containing all hashed passwords with the same salt**:

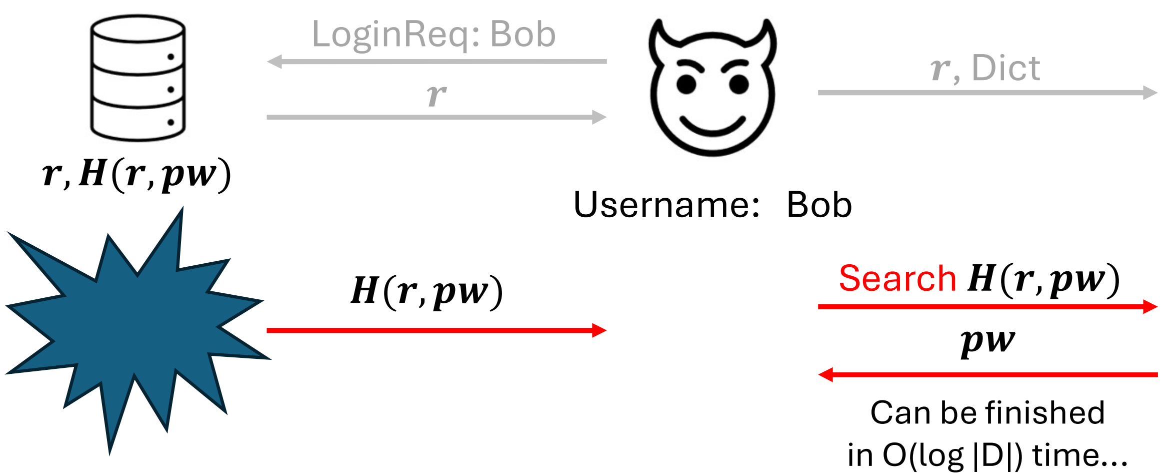


$pw \in$ Dict	The H(pw, r) values
pw_1	$\mathbf{H}(pw_1, r)$
pw_2	$\mathbf{H}(pw_2, r)$
pw_3	$\mathbf{H}(pw_3, r)$
pw_4	$\mathbf{H}(pw_4, r)$
…	…

The table can be computed locally…

Precomputation Attacks on Passwords

- Suppose that the password is stored by hashing and salting.
  - The adversary can learn the salt in some easy ways…
  - **Precompute a table containing all hashed passwords with the same salt**:

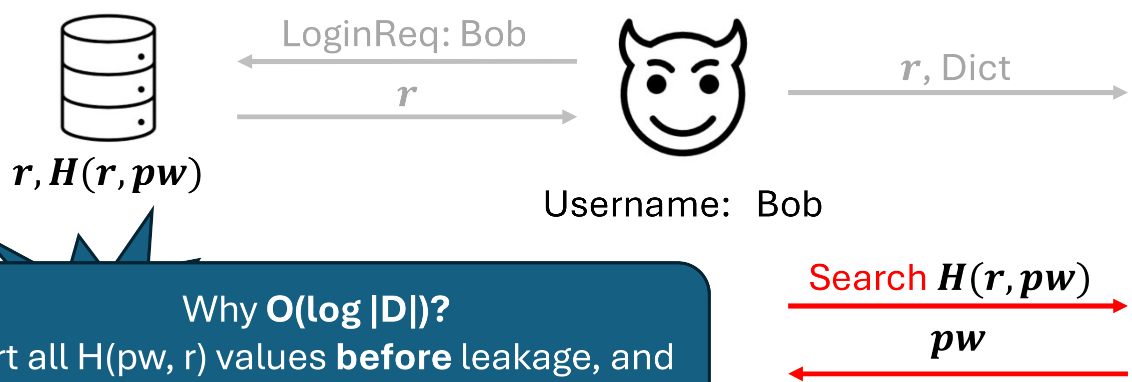


pw ∈ Dict	The H(pw, r) values
pw_1	$\mathbf{H}(pw_1, r)$
pw_2	$\mathbf{H}(pw_2, r)$
pw_3	$\mathbf{H}(pw_3, r)$
pw_4	$\mathbf{H}(pw_4, r)$
…	…

The table can be computed locally…

Precomputation Attacks on Passwords

- Suppose that the password is stored by hashing and salting.
  - The adversary can learn the salt in some easy ways...
  - **Precompute a table containing all hashed passwords with the same salt**:



$pw \in$ Dict	**The H(pw, r) values**
pw_1	$\mathbf{H}(pw_1, r)$
pw_2	$\mathbf{H}(pw_2, r)$
pw_3	$\mathbf{H}(pw_3, r)$
pw_4	$\mathbf{H}(pw_4, r)$
…	…

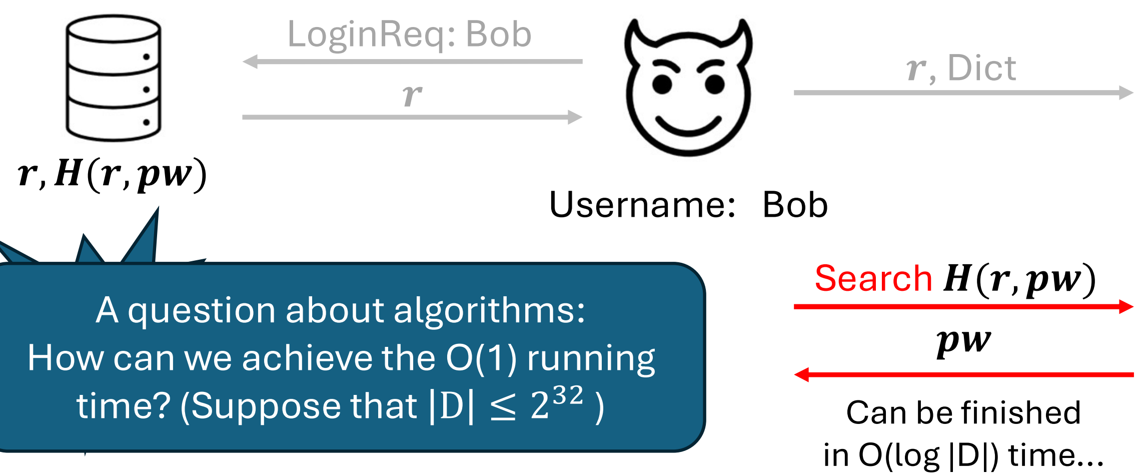
The table can be computed locally...

Can be finished in **O(log |D|)** time...

Why **O(log |D|)?**
Sort all H(pw, r) values **before** leakage, and then binary search...

Precomputation Attacks on Passwords

- Suppose that the password is stored by hashing and salting.
  - The adversary can learn the salt in some easy ways…
  - **Precompute a table containing all hashed passwords with the same salt**:



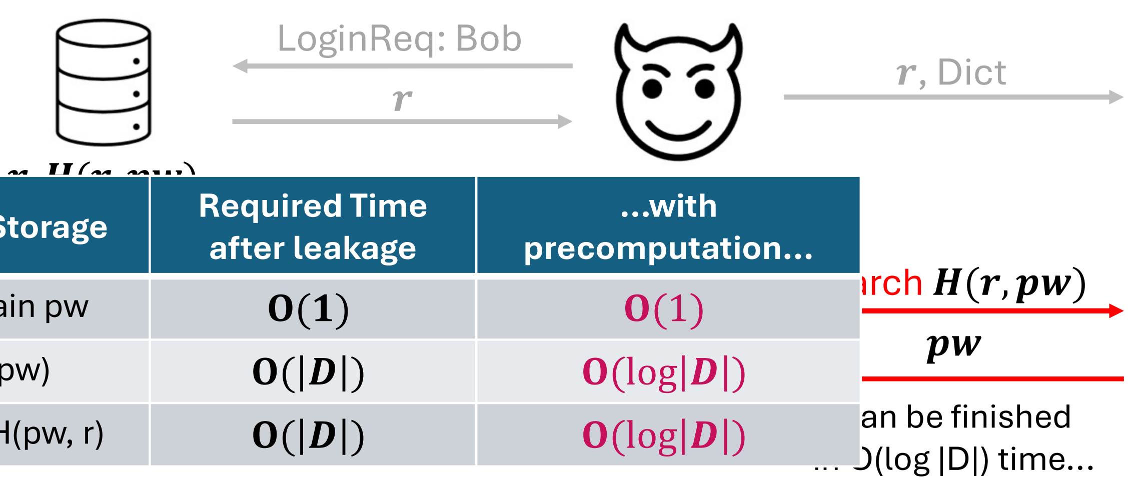
A question about algorithms:
How can we achieve the O(1) running time? (Suppose that $|D| \leq 2^{32}$)

$pw \in$ Dict	**The H(pw, r) values**
pw_1	$\mathbf{H}(pw_1, r)$
pw_2	$\mathbf{H}(pw_2, r)$
pw_3	$\mathbf{H}(pw_3, r)$
pw_4	$\mathbf{H}(pw_4, r)$
…	…

The table can be computed locally…

Precomputation Attacks on Passwords

- Suppose that the password is stored by hashing and salting.
  - The adversary can learn the salt in some easy ways…
  - **Precompute a table containing all hashed passwords with the same salt**:



Storage	Required Time after leakage	…with precomputation…
Plain pw	$O(1)$	$O(1)$
H(pw)	$O(\|D\|)$	$O(\log\|D\|)$
r, H(pw, r)	$O(\|D\|)$	$O(\log\|D\|)$

$pw \in$ Dict	The H(pw, r) values
pw_1	$\mathbf{H}(pw_1, r)$
pw_2	$\mathbf{H}(pw_2, r)$
pw_3	$\mathbf{H}(pw_3, r)$
pw_4	$\mathbf{H}(pw_4, r)$
…	…

The table can be computed locally…

Precomputation Attacks on Passwords

- Comparison:

Attack Method to recover pw	Required Time *before* leakage	**Required Time *after* leakage**
Brute-force on Dictionary	-	$\mathbf{O}(\lvert \boldsymbol{D} \rvert)$
Precomputation	$\leq \mathbf{O}(\lvert \boldsymbol{D} \rvert \cdot \log \lvert \boldsymbol{D} \rvert)$	$\leq \mathbf{O}(\log \lvert \boldsymbol{D} \rvert)$

Precomputation Attacks on Passwords

- Comparison:

Attack Method to recover pw	Required Time *before* leakage	**Required Time *after* leakage**
Brute-force on Dictionary	-	$\mathbf{O}(\lvert \boldsymbol{D} \rvert)$
Precomputation	$\leq \mathbf{O}(\lvert \boldsymbol{D} \rvert \cdot \log \lvert \boldsymbol{D} \rvert)$	$\leq \mathbf{O}(\log \lvert \boldsymbol{D} \rvert)$

- Reveal salt during the protocol => Precomputation attacks
- How can we protect the salt?

Precomputation Attacks on Passwords

- Comparison:

Attack Method to recover pw	Required Time *before* leakage	**Required Time *after* leakage**
Brute-force on Dictionary	-	$\mathbf{O}(\lvert \boldsymbol{D} \rvert)$
Precomputation	$\leq \mathbf{O}(\lvert \boldsymbol{D} \rvert \cdot \log \lvert \boldsymbol{D} \rvert)$	$\leq \mathbf{O}(\log \lvert \boldsymbol{D} \rvert)$

- Reveal salt during the protocol => Precomputation attacks
- How can we protect the salt?
  - No straight-forward solutions that without using algebraic structures
  - Solution using algebraic structures: **Oblivious Pseudorandom Function** (OPRF)

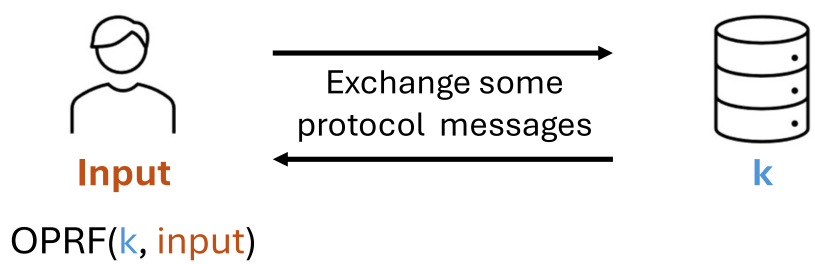
- PAKE without revealing salt: **OPAQUE**

DH-based OPRF

- Classical PRF:
  - Pseudorandomness: If the PRF key is random, then the output of PRF is pseudorandom

DH-based OPRF

- Classical PRF:
  - Pseudorandomness: If the PRF key is random, then the output of PRF is pseudorandom
- Oblivious PRF:
  - Pseudorandomness
  - PRF in the two-party (client-server) computation setting

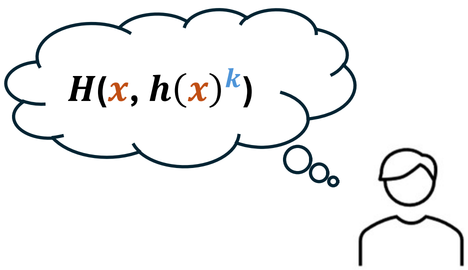


DH-based OPRF

- Classical PRF:
  - Pseudorandomness: If the PRF key is random, then the output of PRF is pseudorandom

- Oblivious PRF:
  - Pseudorandomness
  - PRF in the two-party (client-server) computation setting
  - **Key privacy:** The client learns OPRF(k, input), **but it learns nothing about the key k**
  - **Input privacy:** The server knows the client has evaluated the ORRF, **but it does not know the input**

DH-based OPRF



Input: x

$(\mathbb{G}, g, q)$:

A q-order group $\mathbb{G}$ with a generator g

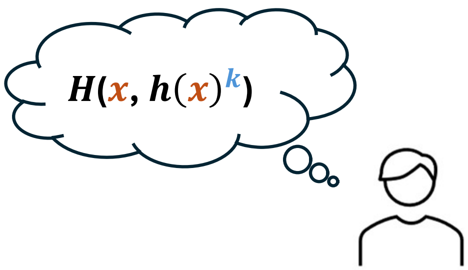
$h: \{0,1\}^* \rightarrow \mathbb{G}$

A hash function map the input into a group element

H: A normal hash function (e.g., SHA256,..)

k

DH-based OPRF



$(\mathbb{G}, g, q)$:

A q-order group $\mathbb{G}$ with a generator g

$h: \{0,1\}^* \rightarrow \mathbb{G}$

A hash function map the input into a group element

H: A normal hash function (e.g., SHA256,..)

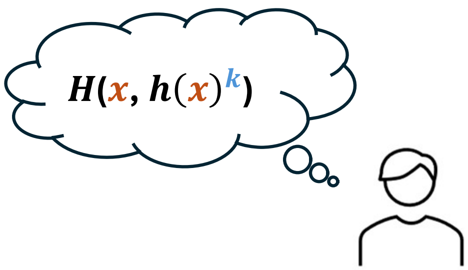
Input: x

k

$\alpha \leftarrow_{\$} \mathbb{Z}_q$

$h(x)^{\alpha}$ →

DH-based OPRF



$(\mathbb{G}, g, q)$:

A q-order group $\mathbb{G}$ with a generator g

$h: \{0,1\}^* \to \mathbb{G}$

A hash function map the input into a group element

H: A normal hash function (e.g., SHA256,..)

Input: x

k

$\alpha \leftarrow_\$ \mathbb{Z}_q$

$h(x)^\alpha$ →

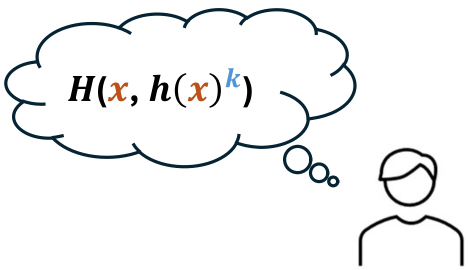
← $h(x)^{\alpha \cdot k}$ $(= (h(x)^\alpha)^k)$

Compute $\alpha^{-1} \in \mathbb{Z}_q$

$h(x)^k = \left(h(x)^{\alpha \cdot k}\right)^{\alpha^{-1}}$

Compute $H(x, h(x)^k)$

DH-based OPRF



$(\mathbb{G}, g, q)$:

A q-order group $\mathbb{G}$ with a generator g

$h: \{0,1\}^* \rightarrow \mathbb{G}$

A hash function map the input into a group element

H: A normal hash function (e.g., SHA256,..)

Input: x

k

$\alpha \leftarrow_{\$} \mathbb{Z}_q$

$h(x)^{\alpha}$ →

← $h(x)^{\alpha \cdot k}$ $(= (h(x)^{\alpha})^{k})$

Key Privacy: $h(x)^k$
=> k, solve dlog...

Input Privacy:
$h(x)^{\alpha}$ is "random"...

Compute $\alpha^{-1} \in \mathbb{Z}_q$

$h(x)^k = \left(h(x)^{\alpha \cdot k}\right)^{\alpha^{-1}}$

Compute $H(x, h(x)^k)$

DH-based OPRF

$(\mathbb{G}, g, q)$:

A q-order group $\mathbb{G}$ with a generator g

$h: \{0,1\}^* \to \mathbb{G}$

A hash function map the input into a group element

H: A normal hash function (e.g., SHA256,..)

Input: $\boldsymbol{x}$

$\boldsymbol{k}$

$\alpha \leftarrow_\$ \mathbb{Z}_q$

$h(\boldsymbol{x})^\alpha$ →

← $h(\boldsymbol{x})^{\alpha \cdot \boldsymbol{k}}$ $(= (h(x)^\alpha)^k)$

Compute $\alpha^{-1} \in \mathbb{Z}_q$

$h(\boldsymbol{x})^{\boldsymbol{k}} = \left(h(\boldsymbol{x})^{\alpha \cdot \boldsymbol{k}}\right)^{\alpha^{-1}}$

Compute $H(x, h(x)^k)$

The OPRF here is
OPRF(*key*: k, *input*: x) = $H(x, h(x)^k)$

DH-based OPRF

$(\mathbb{G}, g, q)$:

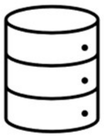
A q-order group $\mathbb{G}$ with a generator g

$h: \{0,1\}^* \rightarrow \mathbb{G}$

A hash function map the input into a group element

H: A normal hash function (e.g., SHA256,..)

Input: $\boldsymbol{pw}$



Key: $\boldsymbol{s}$ (as the salt)

$\alpha \leftarrow_{\$} \mathbb{Z}_q$

$h(\boldsymbol{pw})^{\alpha}$ →

← $h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}}$ $(= (h(x)^{\alpha})^{k})$

Compute $\alpha^{-1} \in \mathbb{Z}_q$

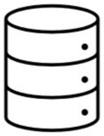
$h(\boldsymbol{pw})^{\boldsymbol{s}} = (h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}})^{\alpha^{-1}}$

Compute $H(\boldsymbol{pw}, h(\boldsymbol{pw})^{\boldsymbol{s}})$

DH-based OPRF

Input: $\boldsymbol{pw}$

$\alpha \leftarrow_\$ \mathbb{Z}_q$



Key: $\boldsymbol{s}$ (as the salt)

$h(\boldsymbol{pw})^{\alpha}$ →

← $h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}}$

$$h(\boldsymbol{pw})^{\boldsymbol{s}} = \left(h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}}\right)^{\alpha^{-1}}$$
$$\boldsymbol{rw} = H(\boldsymbol{pw}, h(\boldsymbol{pw})^{\boldsymbol{s}})$$

- Only the client knows the password
- Only the server knows the salt

DH-based OPRF

Input: $\boldsymbol{pw}$

Key: $\boldsymbol{s}$ (as the salt)

$\alpha \leftarrow_{\$} \mathbb{Z}_q$

$h(\boldsymbol{pw})^{\alpha}$ →

← $h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}}$

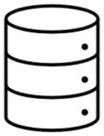
$$h(\boldsymbol{pw})^{\boldsymbol{s}} = (h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}})^{\alpha^{-1}}$$
$$\boldsymbol{rw} = H(\boldsymbol{pw}, h(\boldsymbol{pw})^{\boldsymbol{s}})$$

- Only the client knows the password

- The $\boldsymbol{rw}$ value is pseudorandom by the pseudorandomness of OPRF, **but it can not be directly used as the session key!**
  - rw is always the same, but we expect that a new execution of the protocol produces a new session key...

DH-based OPRF

Input: pw



Key: s (as the salt)

$\alpha \leftarrow_{\$} \mathbb{Z}_q$

$h(pw)^{\alpha}$ →

← $h(pw)^{\alpha \cdot s}$

$$h(pw)^{s} = (h(pw)^{\alpha \cdot s})^{\alpha^{-1}}$$
$$rw = H(pw, h(pw)^{s})$$

- Only the client knows the password

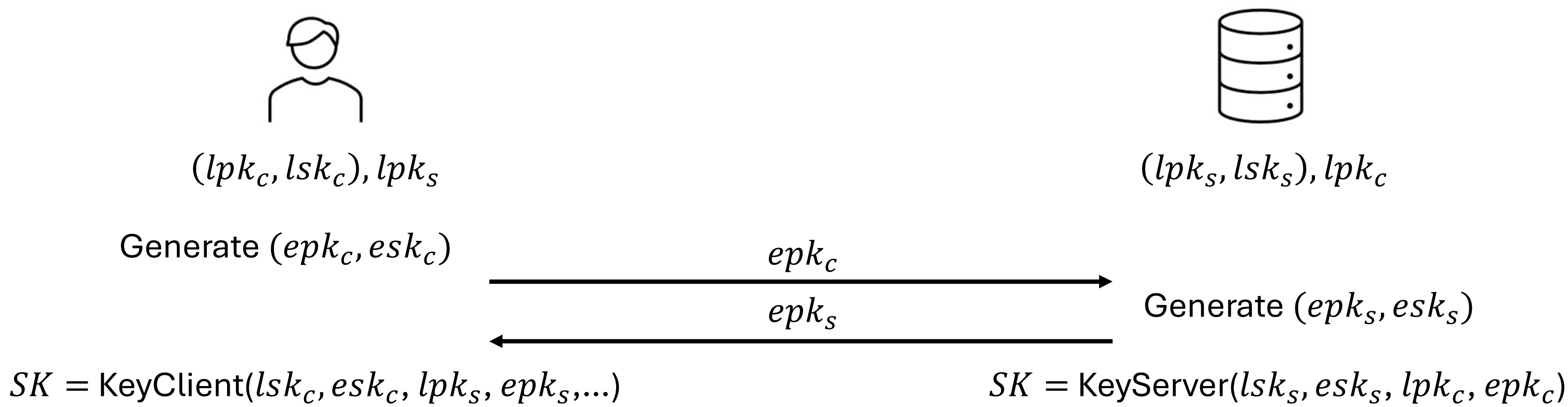
- The rw value is pseudorandom by the pseudorandomness of OPRF, but it can not be directly used as the session key!
  - rw is always the same, but we expect that a new execution of the protocol produces a new session key...
- **Solution: Use AKE protocol to share a session key, and use rw to protect the AKE messages...**

DH-based OPRF + AKE

- Brief introduction of AKE (Authenticated Key Exchange)
  - Two parties share an authenticated key using their long-term key pairs

DH-based OPRF + AKE

- Brief introduction of AKE (Authenticated Key Exchange)
  - Two parties share an authenticated key using their long-term key pairs
  - For example:



  - Security Requirement: Pseudorandom session key, authentication, …

DH-based OPRF + AKE

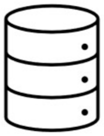
pw

$s, rw = H(pw, h(pw)^{s})$

Suppose that the server has the rw value

DH-based OPRF + AKE

pw



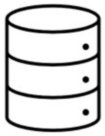
$s, rw = H(pw, h(pw)^s)$

$(lpk_c, lsk_c) \leftarrow$ AKE.KeyGen
$(lpk_s, lsk_s) \leftarrow$ AKE.KeyGen

Generate AKE key pairs

DH-based OPRF + AKE

pw



$s, rw = H(pw, h(pw)^s)$

$(lpk_c, lsk_c) \leftarrow$ AKE.KeyGen
$(lpk_s, lsk_s) \leftarrow$ AKE.KeyGen

key_info = (lpk_c, lsk_c, lpk_s)

rw_key = KDF(rw)
enc_keys = AEAD(rw_key, key_info)

Encrypt generated keys using rw

DH-based OPRF + AKE

pw

$s, rw = H(pw, h(pw)^s)$

$(lpk_c, lsk_c), (lpk_s, lsk_s)$

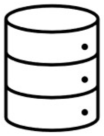
key_info = (lpk_c, lsk_c, lpk_s)

rw_key = KDF(rw)

enc_keys = AEAD(rw_key, key_info)

DH-based OPRF + AKE

pw



s, rw = $H(pw, h(pw)^s)$

$(lpk_c, lsk_c), (lpk_s, lsk_s)$

key_info = (lpk_c, lsk_c, lpk_s)

rw_key = KDF(rw)

enc_keys = AEAD(rw_key, key_info)

$\alpha \leftarrow_\$ \mathbb{Z}_q$

$h(pw)^\alpha$ →

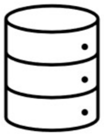
← $h(pw)^{\alpha \cdot s}$

$h(pw)^s = (h(pw)^{\alpha \cdot s})^{\alpha^{-1}}$

$rw = H(pw, h(pw)^s)$

DH-based OPRF + AKE

pw



s, $rw = H(pw, h(pw)^s)$

$(lpk_c, lsk_c), (lpk_s, lsk_s)$

key_info = (lpk_c, lsk_c, lpk_s)

rw_key = KDF(rw)

enc_keys = AEAD(rw_key, key_info)

$\alpha \leftarrow_\$ \mathbb{Z}_q$

$h(pw)^\alpha$ →

← $h(pw)^{\alpha \cdot s}$, enc_keys

$h(pw)^s = (h(pw)^{\alpha \cdot s})^{\alpha^{-1}}$

$rw = H(pw, h(pw)^s)$

rw_key = KDF(rw)

key_info= AEAD.Dec(rw_key, enc_keys) // Client gets (lpk_c, lsk_c, lpk_s)

DH-based OPRF + AKE

pw

s, $rw = H(pw, h(pw)^s)$

$(lpk_c, lsk_c), (lpk_s, lsk_s)$

key_info = (lpk_c, lsk_c, lpk_s)

rw_key = KDF(rw)

enc_keys = AEAD(rw_key, key_info)

$\alpha \leftarrow_\$ \mathbb{Z}_q$

$h(pw)^\alpha$ →

← $h(pw)^{\alpha \cdot s}$, enc_keys

$h(pw)^s = (h(pw)^{\alpha \cdot s})^{\alpha^{-1}}$

$rw = H(pw, h(pw)^s)$

rw_key = KDF(rw)

key_info= AEAD.Dec(rw_key, enc_keys) // Client gets (lpk_c, lsk_c, lpk_s)

Now the client can run the AKE protocol with Server

OPQAUE – Overview of Registration

Username
password: $\boldsymbol{pw}$

("Register", Username, $\boldsymbol{pw}$)

Encrypted by TLS

$\boldsymbol{s} \leftarrow_{\$} \mathbb{Z}_q$

$\boldsymbol{rw} = H(\boldsymbol{pw}, h(\boldsymbol{pw})^{\boldsymbol{s}})$

rw_key = KDF($\boldsymbol{rw}$)

$(lpk_c, lsk_c) \leftarrow$ AKE.KeyGen, $(lpk_s, lsk_s) \leftarrow$ AKE.KeyGen

client_key_info = (lpk_c, lsk_c, lpk_s)

enc_client_keys = AEAD(rw_key, client_key_info)

OPQAUE – Overview of Registration

Username
password: pw

(“Register”, Username, pw)

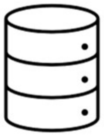
Encrypted by TLS

$s \leftarrow_{\$} \mathbb{Z}_q$

$(lpk_c,$

Then the server store {
user: **Username** // … as index
salt: s
server_k_bundle: lpk_c, lpk_s, lsk_s
client_enc_k_bundle: **enc_client_keys**
… // Auxiliary information
} in the password database

OPQAUE – Stage 1: OPRF



Username, password: ***pw***

$\alpha \leftarrow_{\$} \mathbb{Z}_q$

LoginRequest = (Username, $h(\boldsymbol{pw})^{\alpha}$)

→

OPQAUE – Stage 1: OPRF

Username, password: $\boldsymbol{pw}$

$\alpha \leftarrow_{\$} \mathbb{Z}_q$

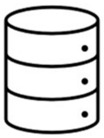
LoginRequest = (Username, $h(\boldsymbol{pw})^{\alpha}$) →

Retrieve (s, server_k_bundle, client_enc_k_bundle)

// ...corresponds to the username

OPQAUE – Stage 1: OPRF

Username, password: $\boldsymbol{pw}$



$\alpha \leftarrow_{\$} \mathbb{Z}_q$

LoginRequest = (Username, $h(\boldsymbol{pw})^{\alpha}$) →

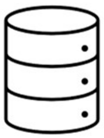
Retrieve ($\boldsymbol{s}$, server_k_bundle, client_enc_k_bundle)
// ...corresponds to the username

← $h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}}$, client_enc_k_bundle

$h(\boldsymbol{pw})^{\boldsymbol{s}} = (h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}})^{\alpha^{-1}}$
$\boldsymbol{rw} = H(\boldsymbol{pw}, h(\boldsymbol{pw})^{\boldsymbol{s}})$
rw_key = KDF($\boldsymbol{rw}$)
client_key_info= AEAD.Dec(rw_key, client_enc_k_bundle)

OPQAUE – Stage 1: OPRF

Username, password: $\boldsymbol{pw}$



$\alpha \leftarrow_{\$} \mathbb{Z}_q$

LoginRequest = (Username, $h(\boldsymbol{pw})^{\alpha}$) →

Retrieve ($\boldsymbol{s}$, server_k_bundle, client_enc_k_bundle)

// ...corresponds to the username

← $h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}}$, client_enc_k_bundle

$h(\boldsymbol{pw})^{\boldsymbol{s}} = (h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}})^{\alpha^{-1}}$

$\boldsymbol{rw} = H(\boldsymbol{pw}, h(\boldsymbol{pw})^{\boldsymbol{s}})$

rw_key = KDF($\boldsymbol{rw}$)

client_key_info= AEAD.Dec(rw_key, client_enc_k_bundle)

Parse client_key_info = (lpk_c, lsk_c, lpk_s)

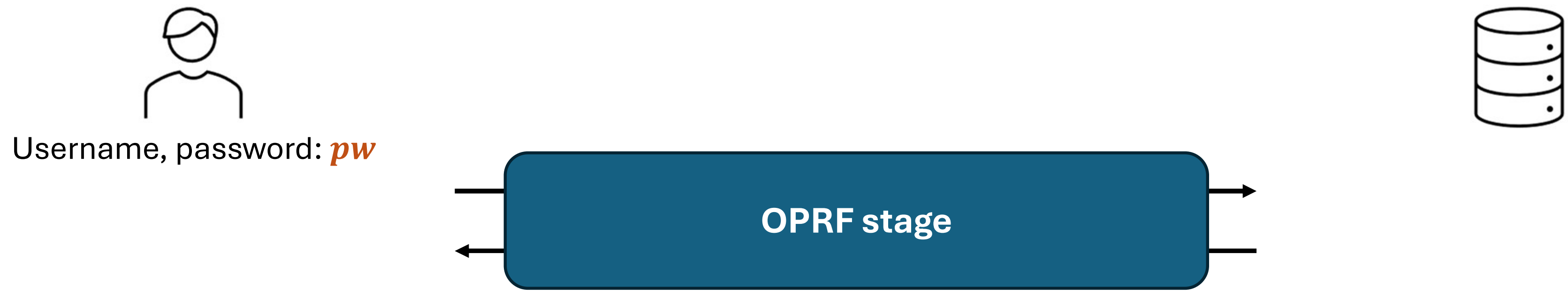
Parse server_k_bundle = (lpk_c, lpk_s, lsk_s)

OPQAUE – Stage 2: AKE

Parse client_key_info = (lpk_c, lsk_c, lpk_s)

Parse server_k_bundle = (lpk_c, lpk_s, lsk_s)

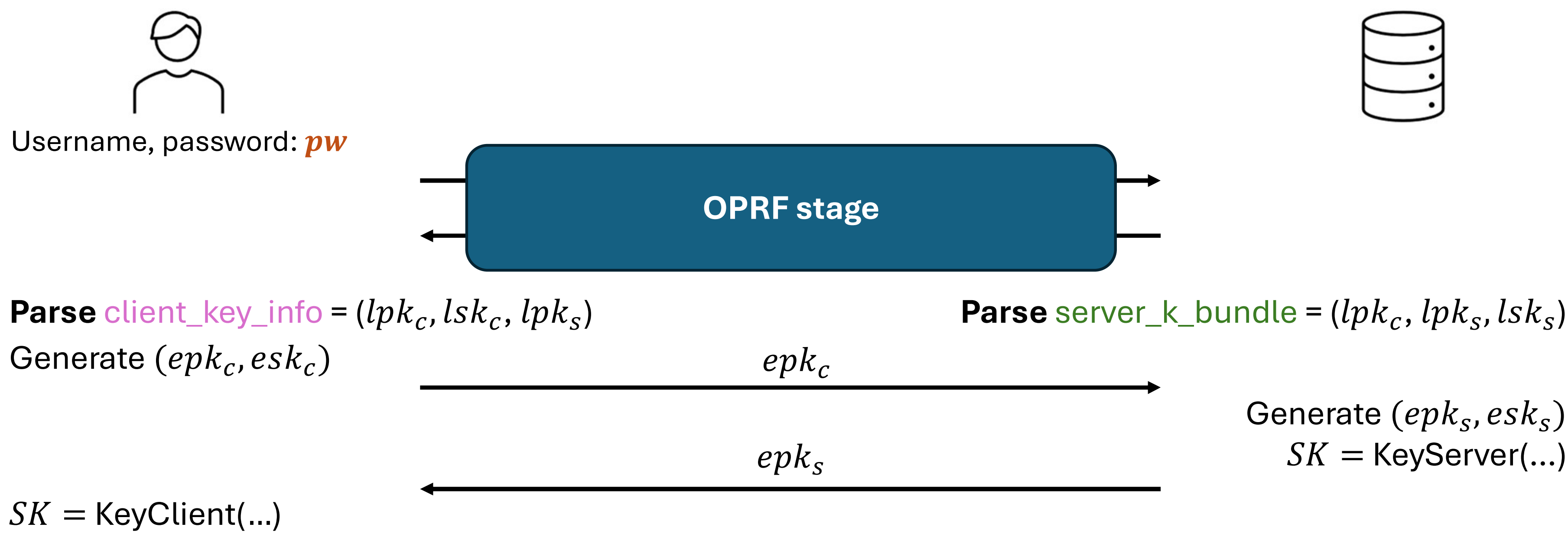
OPQAUE – Stage 2: AKE



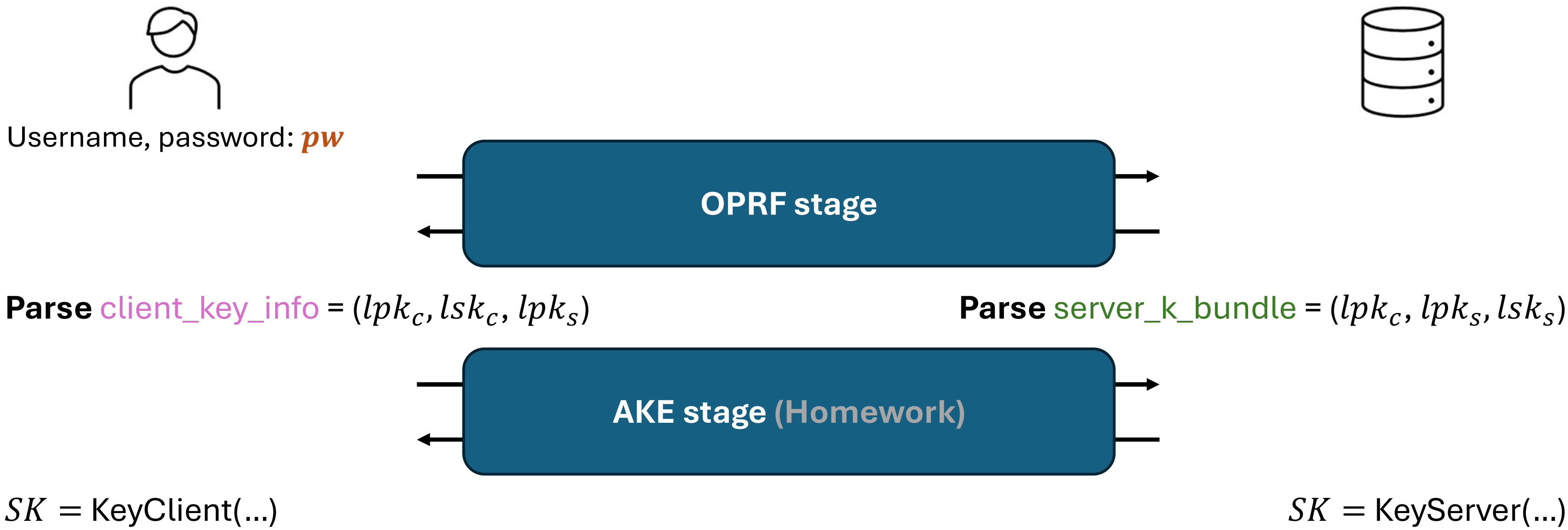
Parse client_key_info = (lpk_c, lsk_c, lpk_s)

Parse server_k_bundle = (lpk_c, lpk_s, lsk_s)

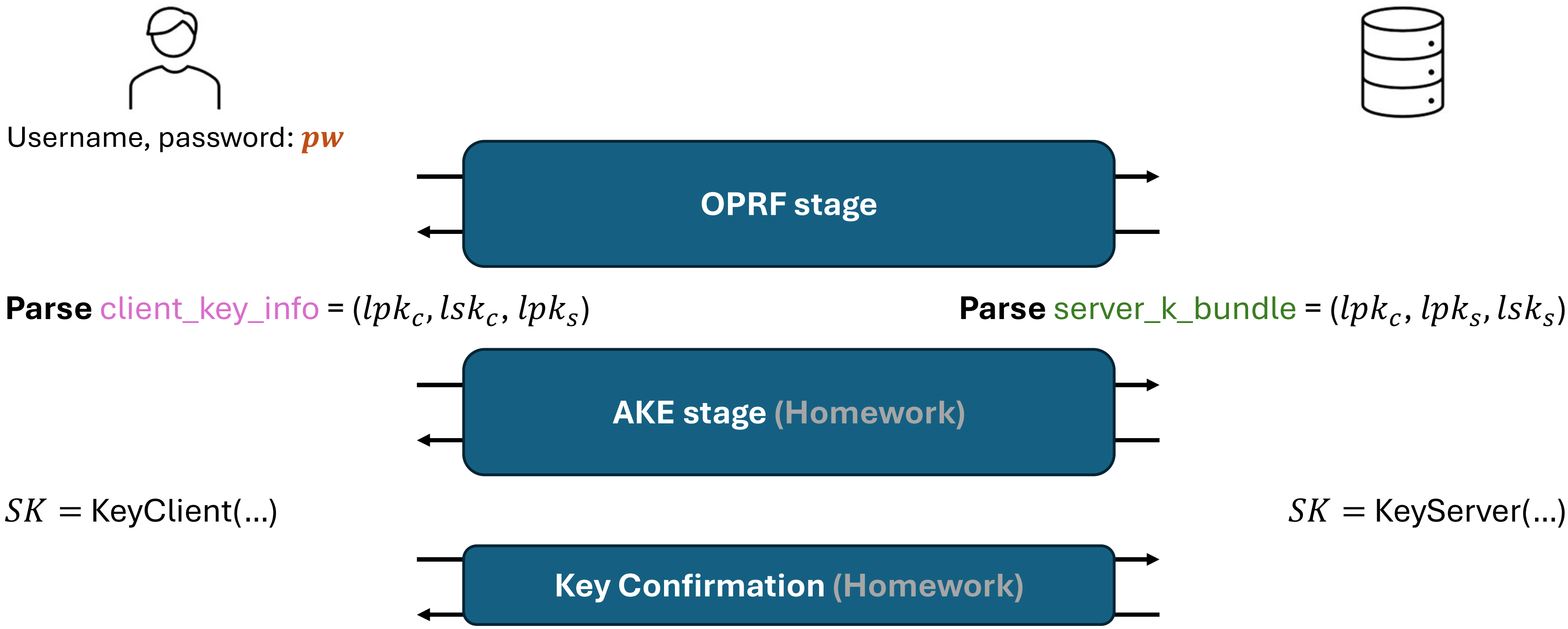
OPQAUE – Stage 2: AKE



OPQAUE – Stage 3: Key Confirmation



OPQAUE – Stage 3: Key Confirmation



OPQAUE – Summary

Username, password: ***pw***

Registration:
Instead of storing (salt, H(salt pw)), we store
(salt, AEAD(rw, [AKE keys], ...)), where rw = DH-OPRF(salt, pw)
// This allows the future messages exchange to not reveal the salt (to prevent precomputation)

OPRF stage:
Allow the client to compute rw (to recover the AKE keys) without revealing the salt

AKE stage:
Use AKE protocol to share a fresh session key

Key Confirmation:
Confirm both parties share the same key

Summary on Password-based Authentication

- Use passwords to authenticate identities

- Storage of passwords & Protocols:
  - Plaintext (or hashed without salt) password: 👎
  - Hashed + salted password: 👍 **(SRP, ...)**
  - Hashed + salted + iterated password: 👍👍 **(SCRAM, ...)**
  - OPRF passwords: 👍👍👍 **(OPAQUE)**

- In Practice: Run over TLS

- Password-based AKE protocols: (secure guarantee even in an insecure TLS connection...)
  - SRP
  - OPAQUE (stronger)

Homework

- Implement the DH-OPRF protocol, and use it to implement the OPAQUE registration phase (using the example code).
- Implement the HMQV AKE protocol

$(\mathbb{G}, g, q)$:

A q-order group $\mathbb{G}$ with a generator g

Client:

$lpk_c = A = g^a \in \mathbb{G}$

$lpk_s = B \in \mathbb{G}$

$lsk_c = a \in \mathbb{Z}_q$

Server:

$lpk_c = A \in \mathbb{G}$

$lpk_s = B = g^b \in \mathbb{G}$

$lsk_s = b \in \mathbb{Z}_q$

HMQV-KG:

1. $lsk \leftarrow_\$ \mathbb{Z}_q$
2. $lpk = g^{lsk}$
3. Return (lpk, lsk)

Client: $x \leftarrow_\$ \mathbb{Z}_q$

Client → Server: $epk_c = x$

Server: $y \leftarrow_\$ \mathbb{Z}_q$

Server: $SK =$ HMQV-KServer (b, y, A, X)

Server → Client: $epk_s = y$

Client: $SK =$ HMQV-KClient(a, x, B, Y)

(The HMQV-Kclient/KServer algorithms are given on the next page...)

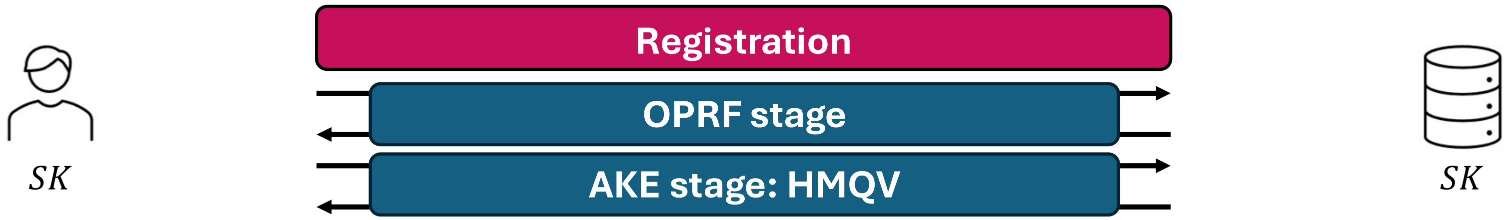
Homework

HMQV-KClient(a, x, B, Y)
1. $d =$ SHA256(X, [Client's Name])
2. $e =$ SHA256(Y, [Server's Name])
3. $ss = (YB^e)^{x+da \bmod q}$
4. $SK =$HKDF(ss)

HMQV-KServer(b, y, A, X)
1. $d =$ SHA256(X, [Client's Name])
2. $e =$ SHA256(Y, [Server's Name])
3. $ss = \left(XA^d\right)^{y+eb \bmod q}$
4. $SK =$HKDF(ss)

Homework

- Implement the OPAQUE protocol instantiating with the HMQV protocol, where the Key Confirmation works as follows:



Client:

$(K_c, K_s) = \text{HKDF}(SK, (\text{“Key Confirmation”})$
$mac_c = \text{HMAC}(K_c, \text{“Client KC”})$
$mac'_s = \text{HMAC}(K_s, \text{“Server KC”})$

Server:

$(K_c, K_s) = \text{HKDF}(SK, (\text{“Key Confirmation”})$
$mac'_c = \text{HMAC}(K_c, \text{“Client KC”})$
$mac_s = \text{HMAC}(K_s, \text{“Server KC”})$

Client → Server: mac_c

Server → Client: mac_s

Client: Check $mac'_s =_? mac_s$
If valid, then accept SK

Server: Check $mac'_c =_? mac_c$
If valid, then accept SK

Homework

- **(Bonus)** Implement the OPAQUE protocol (in the non-bonus homework) using sockets.
- **(Bonus)** What is the RTT of the OPAQUE protocol in the non-bonus homework? Can you improve it? If so, implement your improved version (can be without sockets)
  - One RTT = One "⇄" in the protocol…

Further Reading

- OPAQUE paper: https://eprint.iacr.org/2018/163
- OPAQUE IETF draft: https://www.ietf.org/archive/id/draft-irtf-cfrg-opaque-02.html
- HMQV paper: https://eprint.iacr.org/2005/176

Notes on Homework

- 1 *non-bonus* homework question = 1 point
- 1 *bonus* homework question = 2 points
- How to calculate the final grade of homework (≤ 40):

$$40 \times \left({}^{\text{points you obtain}} \big/ {}_{\text{the number of questions}} \right)$$

// You need to get at least 40 × 60% = **24 points to qualify for the final exam**.

- You can submit **bonus homework before the final deadline: Feb 7th, 2025**
  - Please ensure that your code runs correctly, as you will not have an opportunity to resubmit it.

- If your code for **Homework Set 1 or 2 does not run correctly…**
  - You can resubmit it by the **extended deadline: January 21st, 2025.**

- Some suggestions:
  - **Include the sample input and its expected output** in the README file to help me verify your submission.