Cryptography Engineering

- Lecture 11 (Jan 21, 2026)
- Today's notes:
  - Pre-computation on hashed passwords
  - The OPAQUE protocol
  - Summary on password-based authentication

- Coding tasks/Homework:
  - Offline dictionary attacks
  - Pre-computation attacks v.s. offline attacks without pre-computation
  - Analyze the SCRAM protocol
  - Implement the OPAQUE protocol

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The SCRAM protocol

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The SCRAM 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.
  - Send $\boldsymbol{r}$ to the client, then the client prove its identity by responding $\boldsymbol{H}(\boldsymbol{pw}, \boldsymbol{r})$
  - Encrypted by TLS
- The SCRAM protocol

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.
  - Send $\boldsymbol{r}$ and $\boldsymbol{n}$ to the client, then the client prove its identity by responding $\boldsymbol{H}^{\boldsymbol{n}}(\boldsymbol{pw}, \boldsymbol{r})$
  - The server also needs to prove that it knows $\boldsymbol{H}^{\boldsymbol{n}}(\boldsymbol{pw}, \boldsymbol{r})$
  - Encrypted by TLS
  - Larger n => it takes longer to recover pw

Previous Password-based Protocols

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

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

Previous Password-based Protocols

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

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

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The TLS + SCRAM 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 **required time** to recover the password after leakage using offline attacks

Previous Password-based Protocols

- TLS + hashed & salted passwords
- The TLS + SCRAM 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 **required time** to recover the password after leakage using offline attacks

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

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 **required time** to recover the password after leakage using offline attacks

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

Previous Password-based Protocols

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

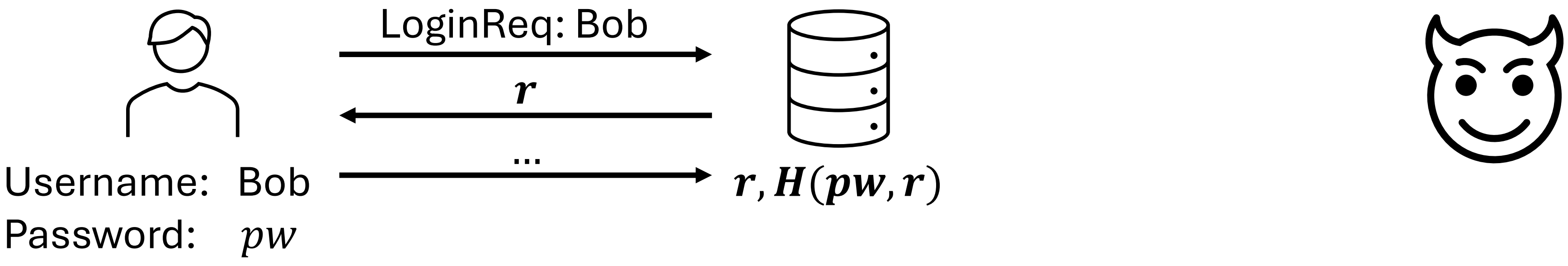
Storage	Required Time after leakage
Plain pw	$\mathbf{O(1)}$
H(pw)	$\mathbf{O}(\|\boldsymbol{D}\|)$
r, H(pw, r)	$\mathbf{O}(\|\boldsymbol{D}\|)$

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

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

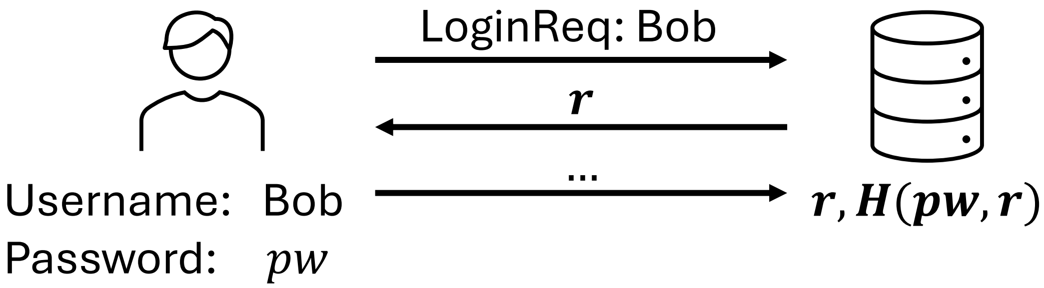
Precomputation Attacks on Passwords

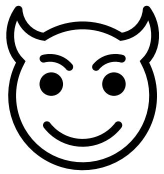
- Suppose that the password is stored by hashing and salting.
  - **The adversary can learn the salt in some easy ways…**



Precomputation Attacks on Passwords

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



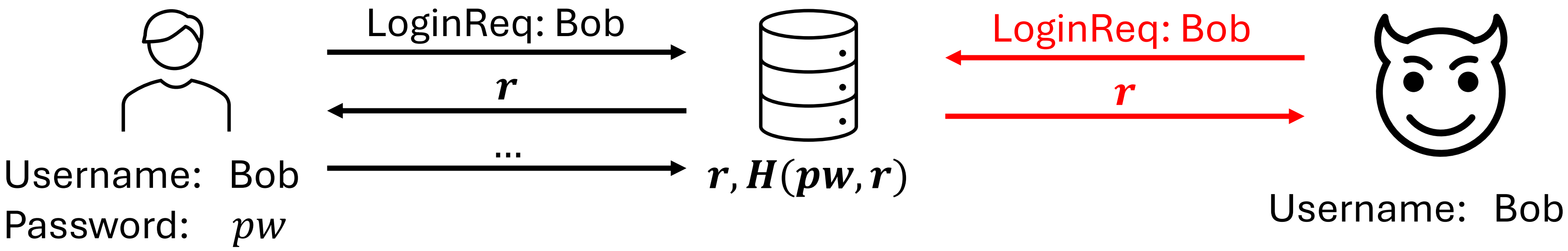


Username: Bob

Suppose that the adversary knows the username…

Precomputation Attacks on Passwords

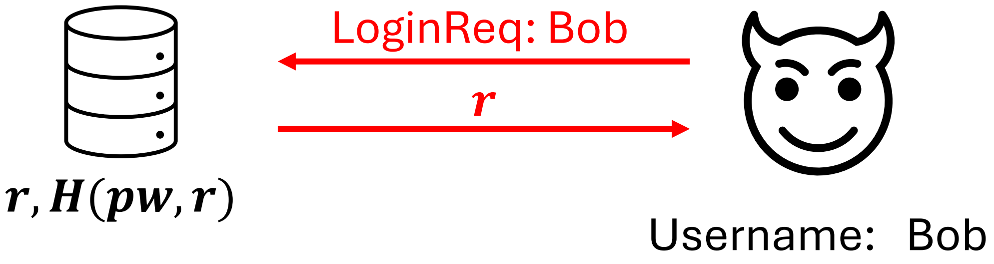
- Suppose that the password is stored by hashing and salting.
  - **The adversary can learn the salt in some easy ways...**



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

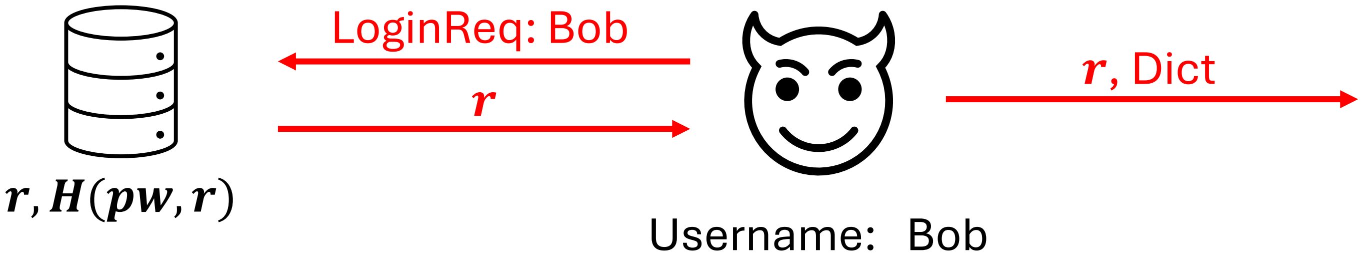
Precomputation Attacks on Passwords

- 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**:

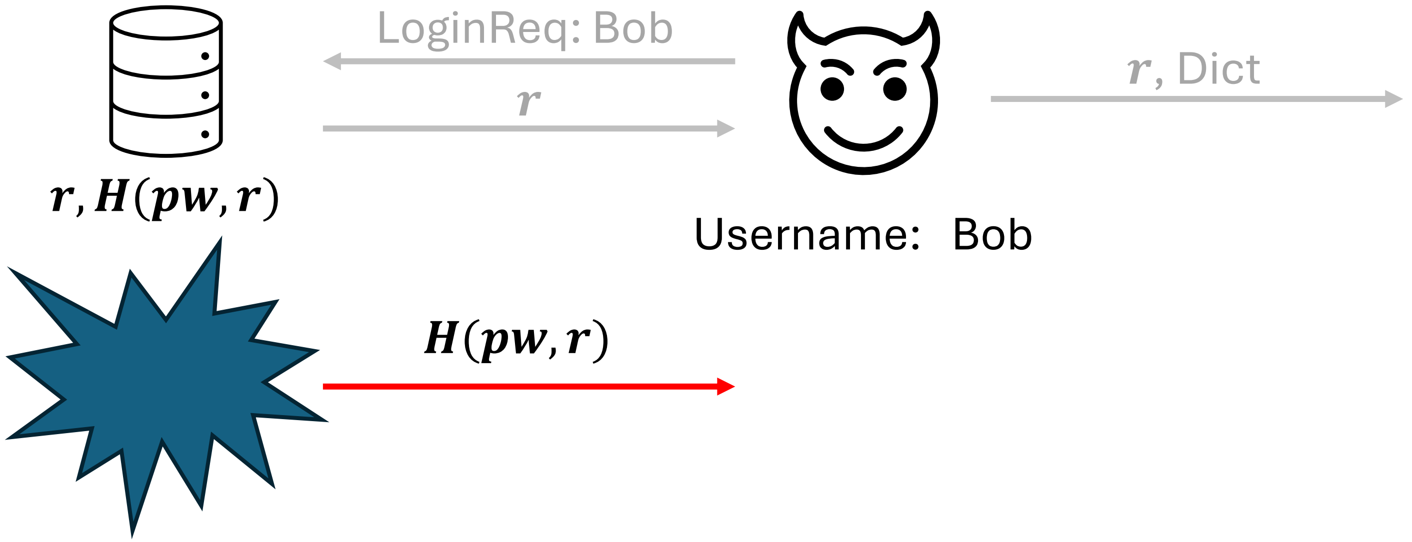


$pw \in$ Dict	**The H(pw, r) values**
pw_1	$\mathbf{H}(pw_1, r)$
pw_2	$\mathbf{H}(pw_2, r)$
pw_3	$\mathbf{H}(pw_3, r)$
pw_4	$\mathbf{H}(pw_4, r)$
…	…

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**:

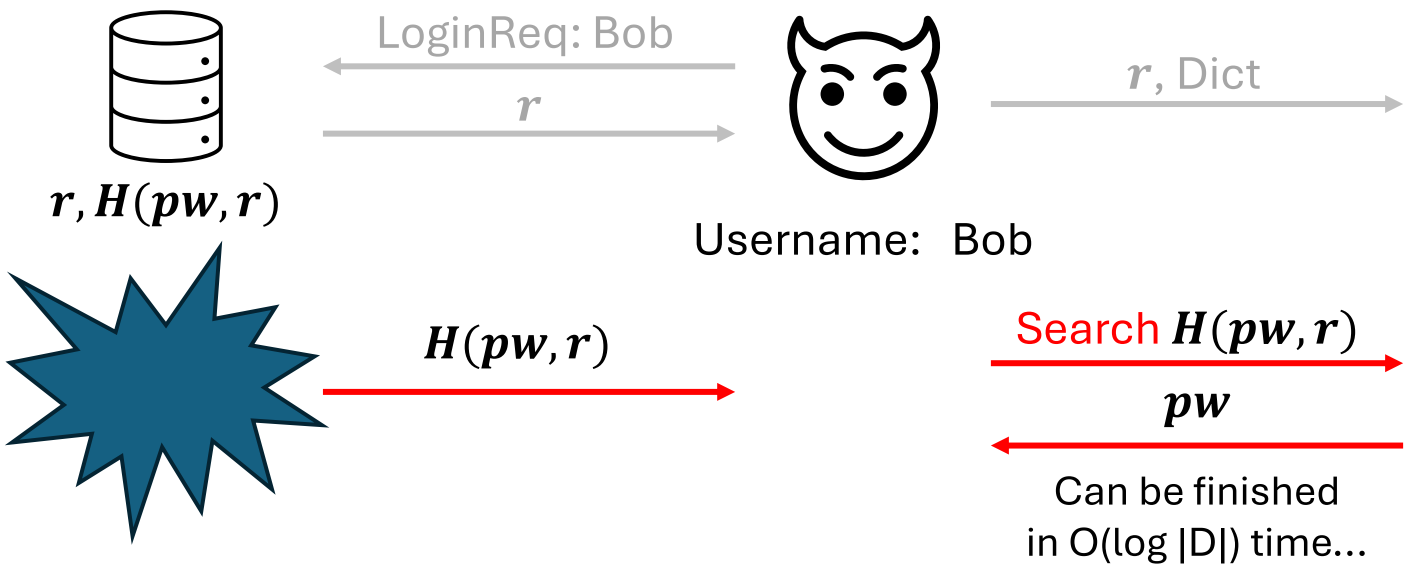


$pw \in$ Dict	The H(pw, r) values
pw_1	$\mathbf{H}(pw_1, r)$
pw_2	$\mathbf{H}(pw_2, r)$
pw_3	$\mathbf{H}(pw_3, r)$
pw_4	$\mathbf{H}(pw_4, r)$
…	…

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:**

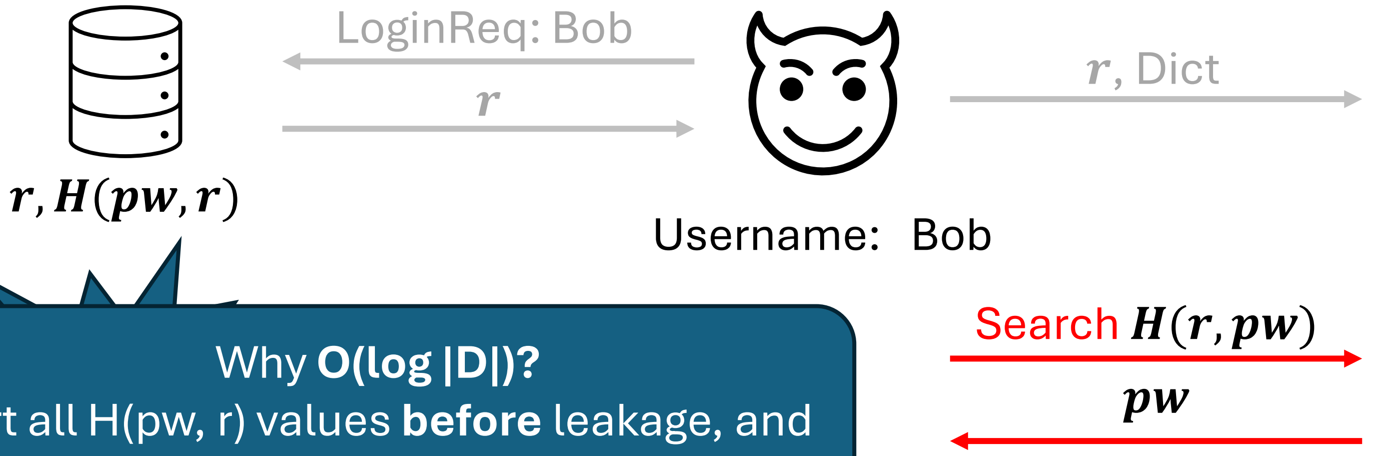


$pw \in$ Dict	The H(pw, r) values
pw_1	$\mathbf{H}(pw_1, r)$
pw_2	$\mathbf{H}(pw_2, r)$
pw_3	$\mathbf{H}(pw_3, r)$
pw_4	$\mathbf{H}(pw_4, r)$
…	…

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:**



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

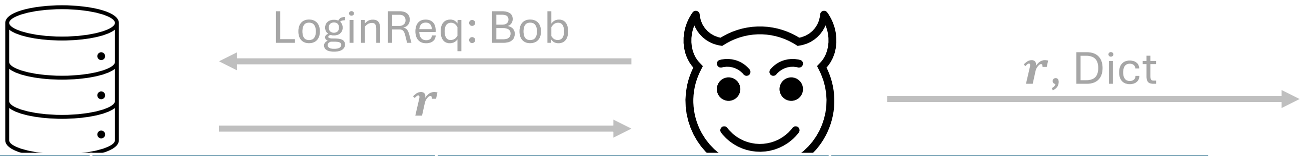
$pw \in$ Dict	**The H(pw, r) values**
pw_1	$\mathbf{H}(pw_1, r)$
pw_2	$\mathbf{H}(pw_2, r)$
pw_3	$\mathbf{H}(pw_3, r)$
pw_4	$\mathbf{H}(pw_4, r)$
…	…

The table can be computed locally...

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**:



Storage	Required Time after leakage	no salt revealed + precomputation	salt revealed + precomputation
Plain pw	$\mathbf{O}(\mathbf{1})$	$\mathbf{O}(1)$	$\mathbf{O}(1)$
H(pw)	$\mathbf{O}(\|\boldsymbol{D}\|)$	$\mathbf{O}(\log\|\boldsymbol{D}\|)$	$\mathbf{O}(\log\|\boldsymbol{D}\|)$
r, H(pw, r)	$\mathbf{O}(\|\boldsymbol{D}\|)$	$\mathbf{O}(\|\boldsymbol{D}\|)$	$\mathbf{O}(\mathbf{log}\|\boldsymbol{D}\|)$

in O(log |D|) time...

$pw \in$ 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 non-cryptographic solutions
  - Cryptographic solution using algebraic structures: **Oblivious Pseudorandom Function** (OPRF)

- Password authentication protocol 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

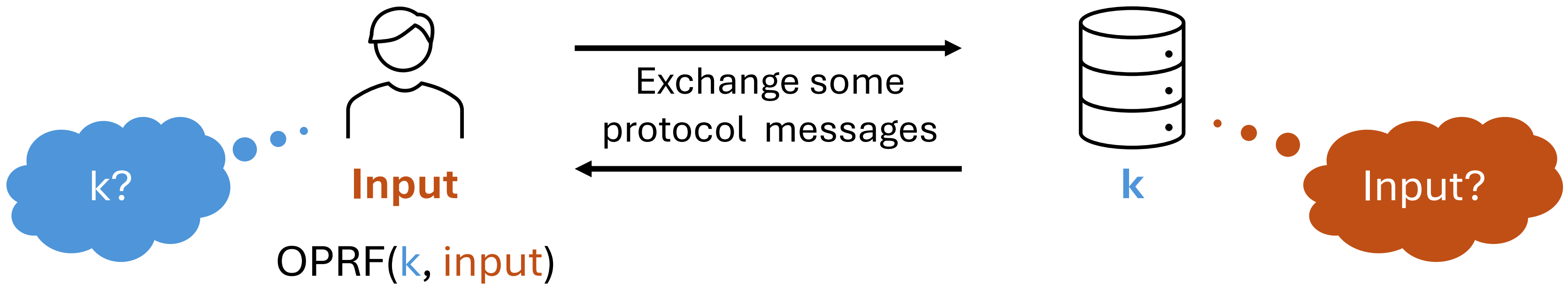
- Oblivious PRF:
  - Pseudorandomness
  - PRF in the two-party (client-server) computation setting

OPRF(k, input)

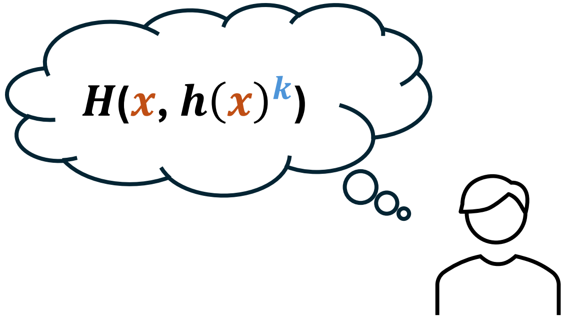
DH-based OPRF

- 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

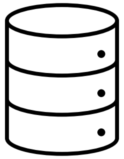
$(\mathbb{G}, g, q)$:

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

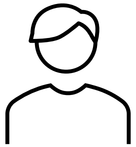
H: A normal hash function (e.g., SHA256,..)



k

DH-based OPRF

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



Input: x

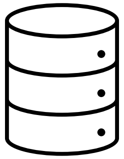
$(\mathbb{G}, g, q)$:

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,..)



k

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

$$\xrightarrow{h(x)^{\alpha}}$$

DH-based OPRF

$\boldsymbol{H}(\boldsymbol{x}, \boldsymbol{h}(\boldsymbol{x})^{\boldsymbol{k}})$

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

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 $H(x, h(x)^{k})$

DH-based OPRF

$\boldsymbol{H(x, h(x)^{k})}$

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

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: $\boldsymbol{x}$

$\boldsymbol{k}$

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

$h(\boldsymbol{x})^{\alpha}$ $\longrightarrow$

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

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

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

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

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

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

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: $\boldsymbol{x}$

$\boldsymbol{k}$

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

$$h(\boldsymbol{x})^{\alpha} \longrightarrow$$

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

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

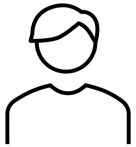
$(\mathbb{G}, g, q)$:

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

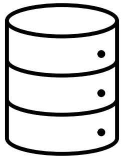
$h: \{0,1\}^* \rightarrow \mathbb{G}$

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)

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

$h(\boldsymbol{pw})^{\alpha}$ $\longrightarrow$

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

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: $\boldsymbol{pw}$

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

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

$$\xrightarrow{\quad h(\boldsymbol{pw})^{\alpha} \quad}$$

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

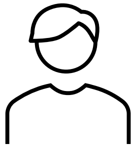
$$h(\boldsymbol{pw})^{\boldsymbol{s}} = \left(h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}}\right)^{\alpha^{-1}}$$

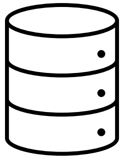
$$\boldsymbol{rw} = H(\boldsymbol{pw}, h(\boldsymbol{pw})^{\boldsymbol{s}})$$

- Only the client knows the password

- Only the server knows the salt

DH-based OPRF





Input: $\boldsymbol{pw}$

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

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

$$h(\boldsymbol{pw})^{\alpha} \longrightarrow$$

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

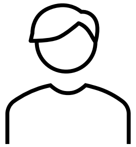
$$h(\boldsymbol{pw})^{\boldsymbol{s}} = \left(h(\boldsymbol{pw})^{\alpha \cdot \boldsymbol{s}}\right)^{\alpha^{-1}}$$

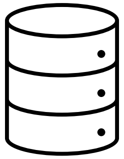
$$\boldsymbol{rw} = H(\boldsymbol{pw}, h(\boldsymbol{pw})^{\boldsymbol{s}})$$

- Only the client knows the password

- 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: $\boldsymbol{pw}$

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

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

$$\xrightarrow{\quad h(\boldsymbol{pw})^{\alpha} \quad}$$

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

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

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

- Only the client knows the password

- 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...
- **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:

Client		Server
$(lpk_c, lsk_c), lpk_s$		$(lpk_s, lsk_s), lpk_c$
Generate (epk_c, esk_c)	$\xrightarrow{epk_c}$	
	$\xleftarrow{epk_s}$	Generate (epk_s, esk_s)
$SK =$ KeyClient$(lsk_c, esk_c, lpk_s, epk_s, \ldots)$		$SK =$ KeyServer$(lsk_s, esk_s, lpk_c, epk_c)$

  - Security Requirement: Pseudorandom session key, authentication, …

DH-based OPRF + AKE

- Brief introduction of AKE (Authenticated Key Exchange)
  - Concrete example: The TripleDH (3DH) protocol

$(lpk_c, lsk_c) = (g^a, a),$
$lpk_s = g^b$

$(lpk_s, lsk_s) = (g^b, b),$
$lpk_c = g^a$

Generate
$(epk_c = g^x, esk_c = x)$

$\xrightarrow{epk_c \,=\, g^x}$

$\xleftarrow{epk_s \,=\, g^y}$

Generate
$(epk_s = g^y, esk_s = y)$

- The session key is SK = HKDF$(g^a, g^b, g^x, g^y, g^{ay}, g^{xb}, g^{xy})$

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)

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

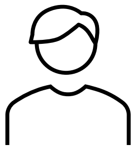
$h(pw)^\alpha$ $\longrightarrow$

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

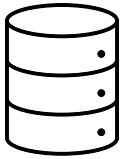
$h(pw)^s = (h(pw)^{\alpha \cdot s})^{\alpha^{-1}}$

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

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)

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

$$\xrightarrow{\quad h(pw)^{\alpha} \quad}$$

$$\xleftarrow{\quad h(pw)^{\alpha \cdot s}, \text{ enc\_keys} \quad}$$

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

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

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**

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

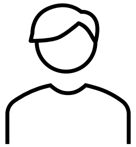
$rw = H(pw, h(pw)^s)$

rw_key = KDF(rw)

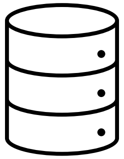
key_info= AEAD.Dec(rw_key, enc_keys) // Client gets (lpk_c, lsk_c, lpk_s)

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

$\boldsymbol{s} \leftarrow_{\$} \mathbb{Z}_q$ //Each user should have unique salt

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

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

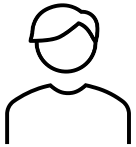
("Register", Username, pw)

Encrypted by TLS

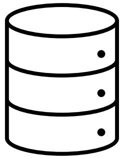
$(lpk_c,$

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

OPQAUE – Stage 1: OPRF



Username, password: $\boldsymbol{pw}$



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

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

OPQAUE – Stage 1: OPRF

Username, password: pw

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

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

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)

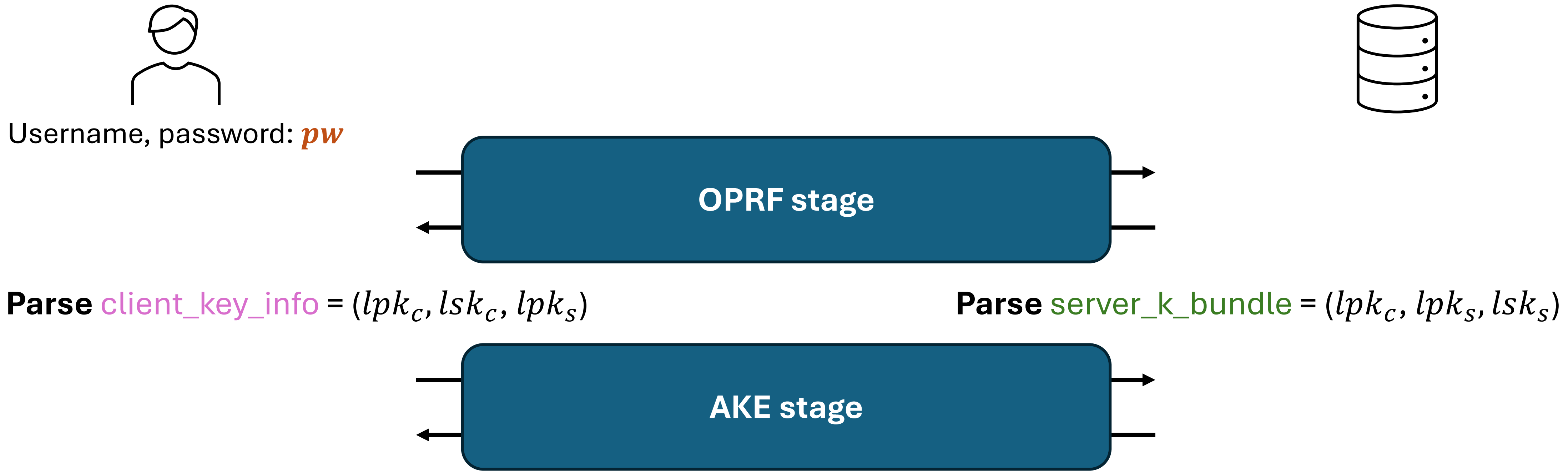
OPQAUE – Stage 2: AKE

Username, password: pw

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 2: AKE

Username, password: $\boldsymbol{pw}$

OPRF stage

Parse client_key_info = (lpk_c, lsk_c, lpk_s)

Generate (epk_c, esk_c)

Parse server_k_bundle = (lpk_c, lpk_s, lsk_s)

$\xrightarrow{epk_c}$

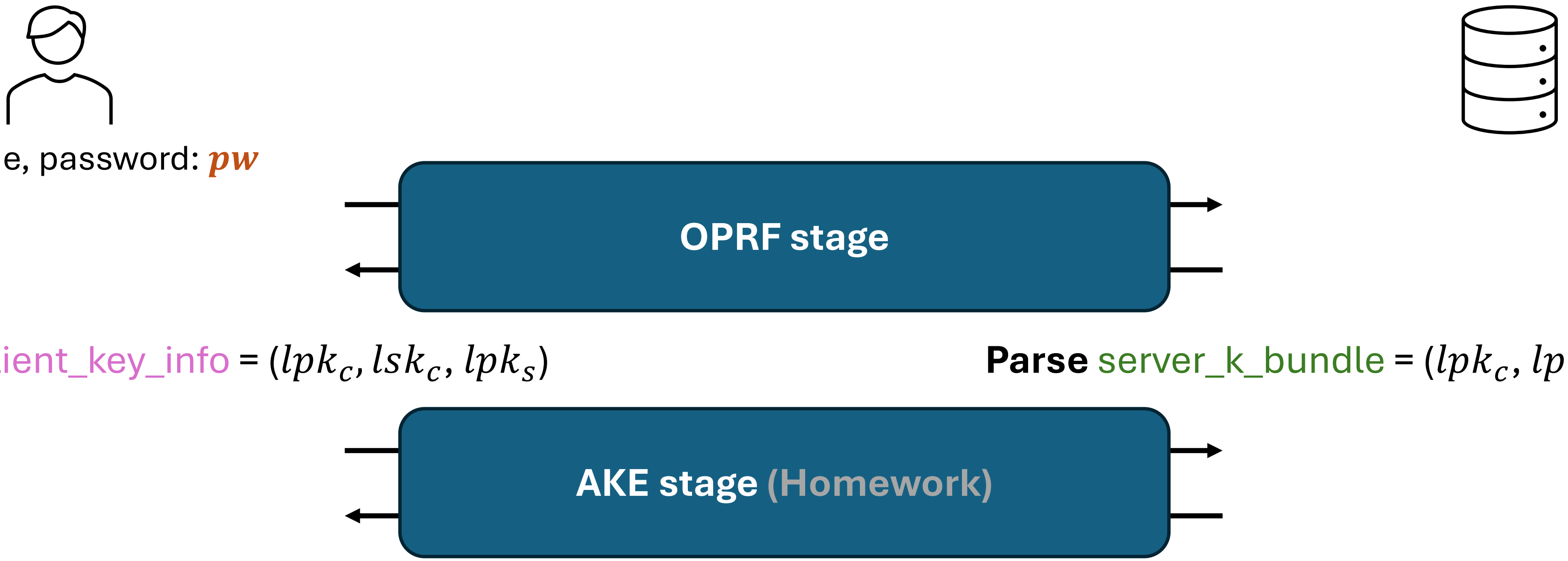
Generate (epk_s, esk_s)

SK = KeyServer(…)

$\xleftarrow{epk_s}$

SK = KeyClient(…)

OPQAUE – Stage 3: Key Confirmation



Username, password: $\boldsymbol{pw}$

Parse client_key_info = (lpk_c, lsk_c, lpk_s)

Parse server_k_bundle = (lpk_c, lpk_s, lsk_s)

SK = KeyClient(…)

SK = KeyServer(…)

OPQAUE – Stage 3: Key Confirmation

Username, password: pw

OPRF stage

Parse client_key_info = (lpk_c, lsk_c, lpk_s)

Parse server_k_bundle = (lpk_c, lpk_s, lsk_s)

AKE stage (Homework)

SK = KeyClient(...)

SK = KeyServer(...)

Key Confirmation (Homework)

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 + iterated password: 👍 **(SCRAM, …)**
  - OPRF passwords: 👍👍 **(OPAQUE)**

- OPAQUE: secure guarantee even in an insecure TLS connection…

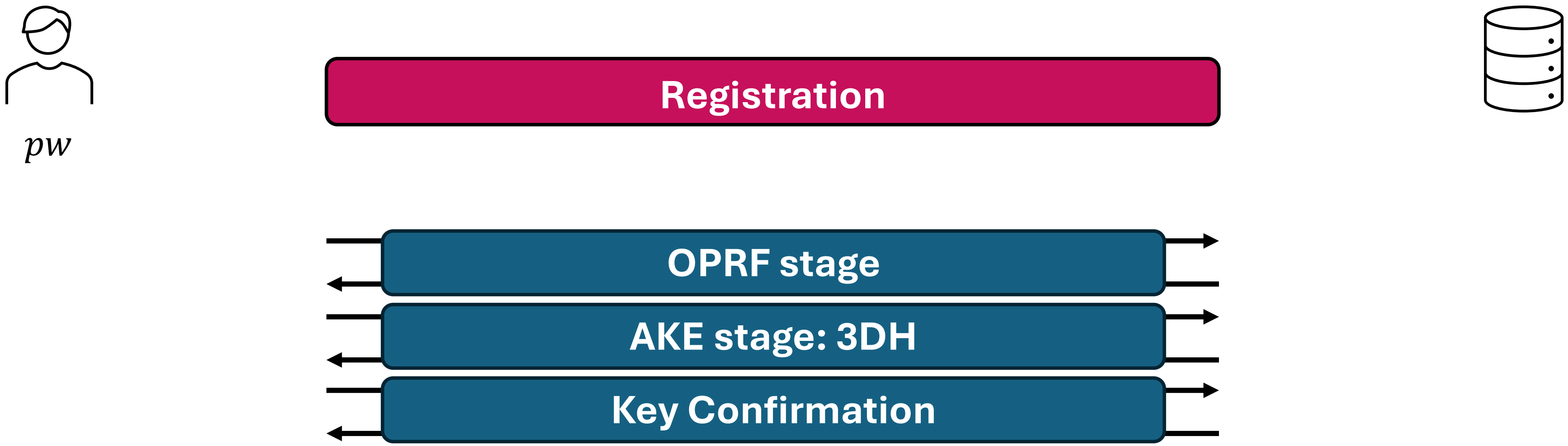
- In Practice: Run over TLS

Homework

- (1) Launch offline attacks on the hashed password (SHA3-256)
  - a. See the sample code https://github.com/RunzhiZeng/CryptoEng_W2526_RustCode
  - b. The target (encoded in base64): 8yQ28QbbPQYfvpta2FBSgsZTGZlFdVYMhn7ePNbaKV8=
  - c. Use SHA3-256 library.
    - Python: hashlib.sha3_256
    - Rust: sha3 = "0.10“, base64 = "0.13"
- (2) Analyze SCRAM (Write a simple pdf document):
  - a. Which parts of SCRAM provide “client authentication”?
  - b. Which parts of SCRAM provide “server authentication”
  - c. If we do not use TLS to protect SCRAM, then which parts may cause offline dictionary attacks?
- (3) Implement the OPAQUE protocol
  - The specification is presented on the next page
  - You need to use hash-to-curve functions when implementing DH-OPRF:
  - Sample code: https://github.com/RunzhiZeng/CryptoEng_W2526_RustCode

Homework

- (3) Implement the following simplified OPAQUE protocol



Homework

AKE stage: 3DH

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

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

Client		Server
$lsk_c = a \in_\$ \mathbb{Z}_q$		$lsk_s = b \in_\$ \mathbb{Z}_q$
$lpk_c = A = g^a \in \mathbb{G}$		$lpk_c = A \in \mathbb{G}$
$lpk_s = B \in \mathbb{G}$		$lpk_s = B = g^b \in \mathbb{G}$
$x \leftarrow_\$ \mathbb{Z}_q$	$\xrightarrow{epk_c = x}$	
		$y \leftarrow_\$ \mathbb{Z}_q$
		$SK =$ 3DH-KServer (b, y, A, X)
	$\xleftarrow{epk_s = y}$	
$SK =$ 3DH-KClient(a, x, B, Y)		

(The 3DH-Kclient/KServer algorithms are given on the next page...)

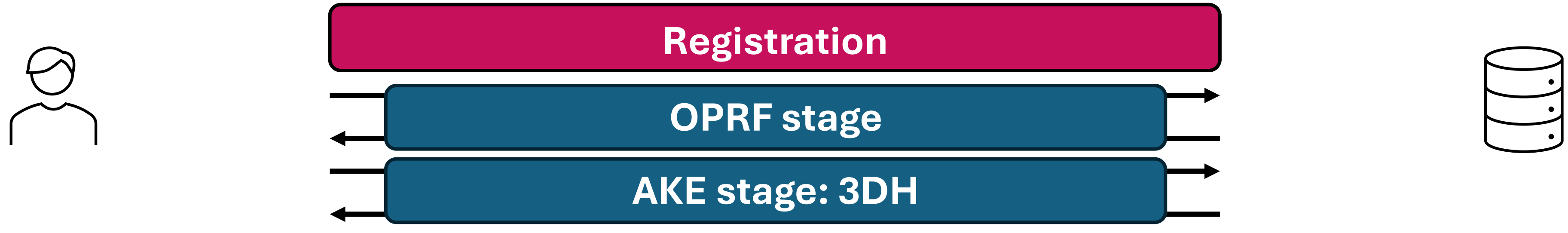
Homework

3DH-KClient(a, x, B, Y)
1. SK =HKDF(B^x, Y^x, Y^a)
2. return SK

3DH-KServer(b, y, A, X)
1. SK =HKDF(X^b, X^y, A^y)
2. return SK

Homework

- (3) Implement the following simplified OPAQUE protocol



Client		Server
SK		SK
$(K_c, K_s) =$ HKDF(SK, (“Key Confirmation”)		$(K_c, K_s) =$ HKDF(SK, (“Key Confirmation”)
$mac_c =$ HMAC(K_c, “Client KC”)		$mac'_c =$ HMAC(K_c, “Client KC”)
$mac'_s =$ HMAC(K_s, “Server KC”)		$mac_s =$ HMAC(K_s, “Server KC”)
	$\xrightarrow{mac_c}$	
	$\xleftarrow{mac_s}$	
Check $mac'_s =_? mac_s$		Check $mac'_c =_? mac_c$
If valid, then accept SK		If valid, then accept SK

Homework

- DDL for 3rd homework set:

Feb 11th , 2026 at 11:59 PM

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