Cryptography Engineering

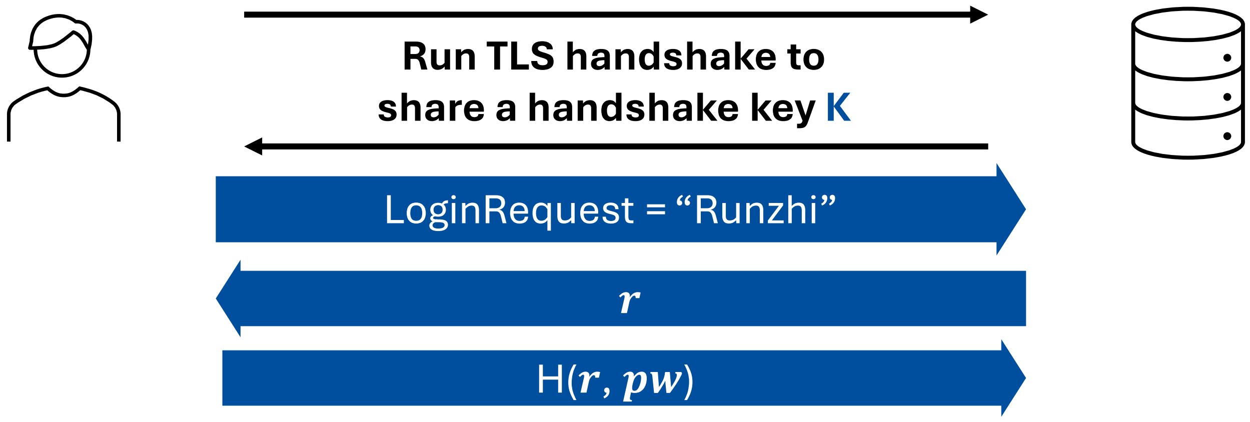
- Lecture 8 (Dec 11, 2024)
- Today's notes:
  - Protocol Study: The SCRAM protocol
  - Password-based Authenticated Key Exchange (PAKE)
  - An (In)secure Example: Encrypted-key-exchange protocol
  - Protocol study: The SRP protocol

- Coding tasks/Homework:
  - Implement the SCRAM protocol
  - **Bonus**: Informal analysis of SRP
  - **Bonus**: Implement pre-computation attacks on SRP

TLS + Salted Hashes of Passwords

- TLS + salted & hashed passwords
  - Use TLS to protect the transmission of pw
  - No TLS handshake key => Cannot launch offline dictionary attacks

Account = “Runzhi”
password = $\boldsymbol{pw}$
where pw is some string



Run TLS handshake to share a handshake key K
LoginRequest = “Runzhi”
$\boldsymbol{r}$
H($\boldsymbol{r}, \boldsymbol{pw}$)

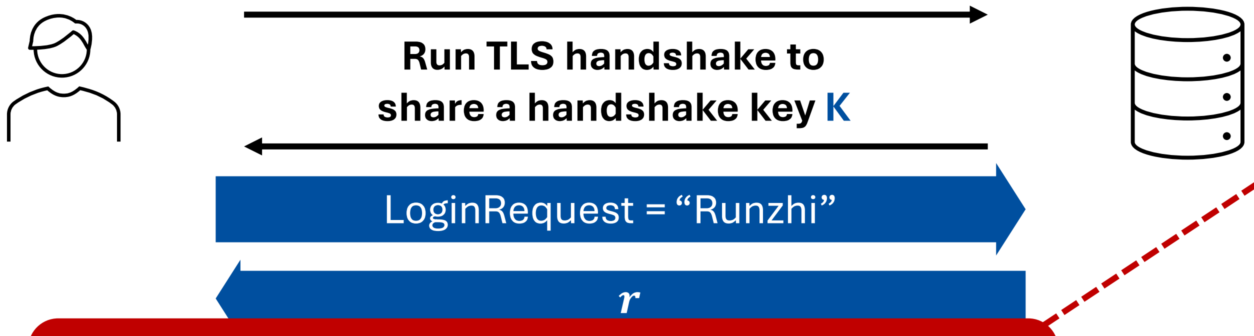
(Encrypted by the TLS handshake key K)

User	password_file
Runzhi	$\boldsymbol{r, H(r, pw)}$
Tom	$r_2, H(r_2, pw_2)$
…	…

TLS + Salted Hashes of Passwords

- TLS + salted & hashed passwords
  - Use TLS to protect the transmission of pw
  - No TLS handshake key => Cannot launch offline dictionary attacks

Account = “Runzhi”
password = pw
where pw is some string



Run TLS handshake to share a handshake key K

LoginRequest = “Runzhi”

r

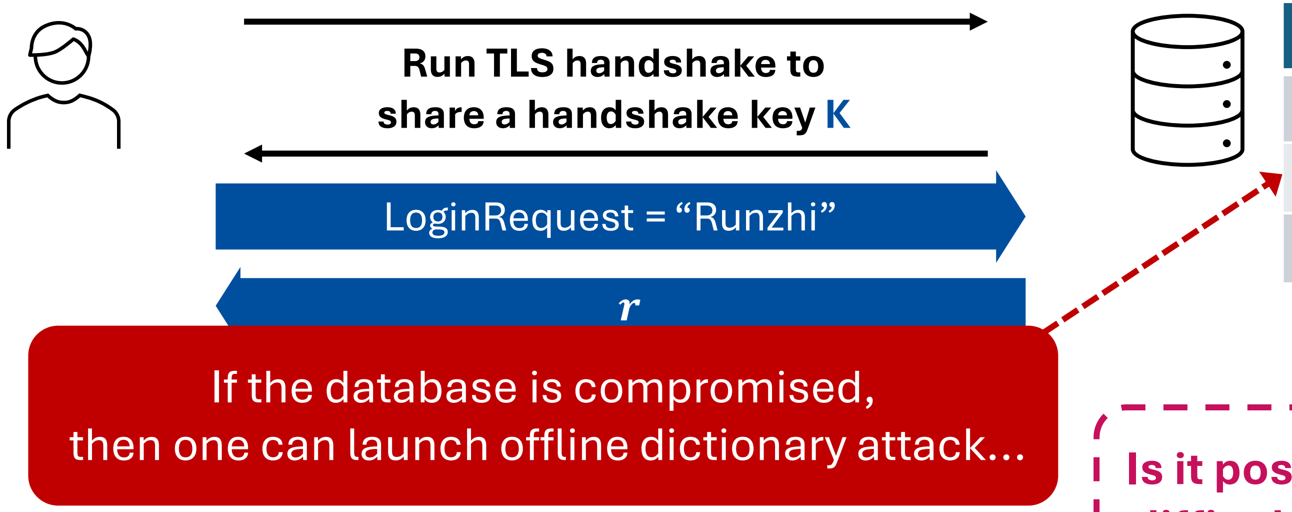
User	password_file
Runzhi	$r, H(r, pw)$
Tom	$r_2, H(r_2, pw_2)$
…	…

If the database is compromised, then one can launch offline dictionary attack…

TLS + Salted Hashes of Passwords

- TLS + salted & hashed passwords
  - Use TLS to protect the transmission of pw
  - No TLS handshake key => Cannot launch offline dictionary attacks

Account = “Runzhi”
password = pw
where pw is some string



If the database is compromised, then one can launch offline dictionary attack…

User	password_file
Runzhi	$\boldsymbol{r, H(r, pw)}$
Tom	$r_2, H(r_2, pw_2)$
…	…

Is it possible to increase the difficulty of offline attacks?

The SCRAM protocol

- Salted Challenge Response Authentication Mechanism

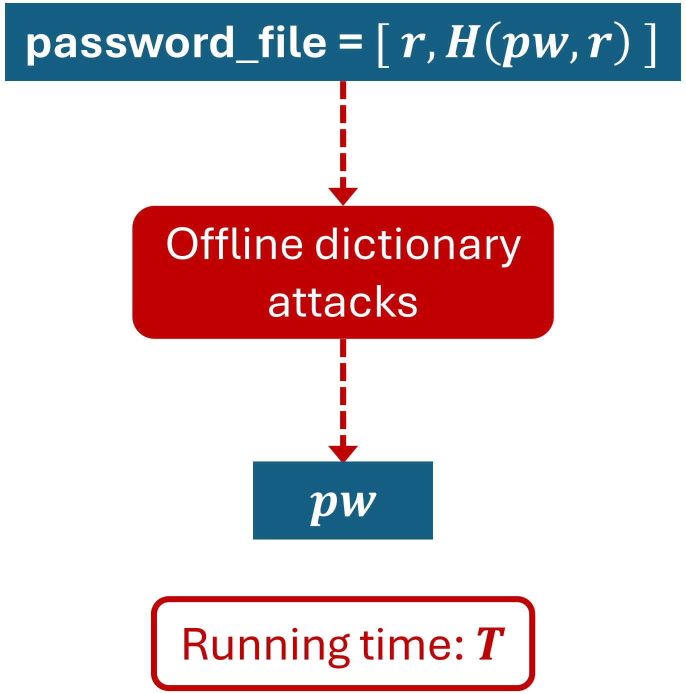
- Main idea:
  1. Add iteration in computing salted & hashed password
  2. Challenge-response Mechanism
  3. Run over TLS

- Other Important Features:
  - ➢ Inherent Resistance to Replay Attacks

    (TLS + salted & hashed passwords resists replay attacks because of TLS, while SCRAM resists replay attacks inherently, independent of the transport layer.)
  - ➢ Mutual Authentication

The SCRAM protocol

- Add iteration in computing salted & hashed password:



password_file = [$r, H(pw, r)$]

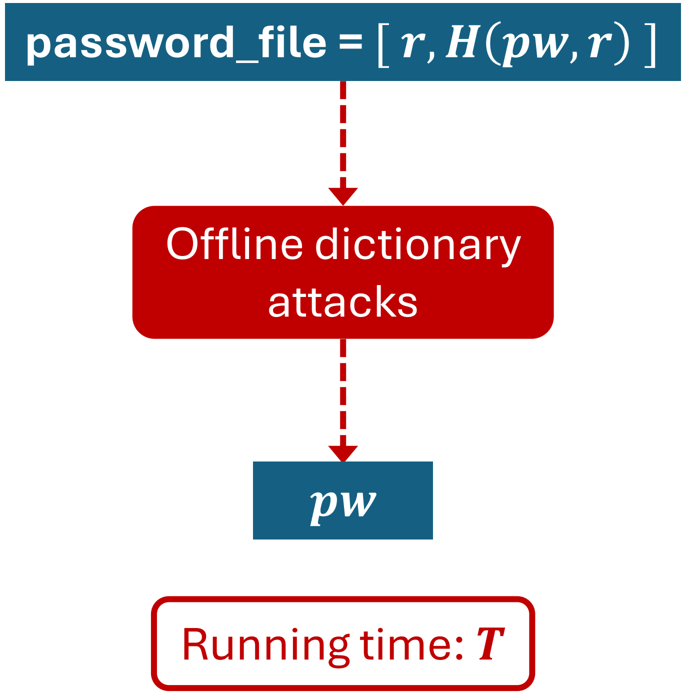
Offline dictionary attacks

pw

Running time: T

The SCRAM protocol

- Add iteration in computing salted & hashed password:



password_file = $[\, \boldsymbol{r}, \boldsymbol{H}(\boldsymbol{pw}, \boldsymbol{r}) \,]$

Offline dictionary attacks

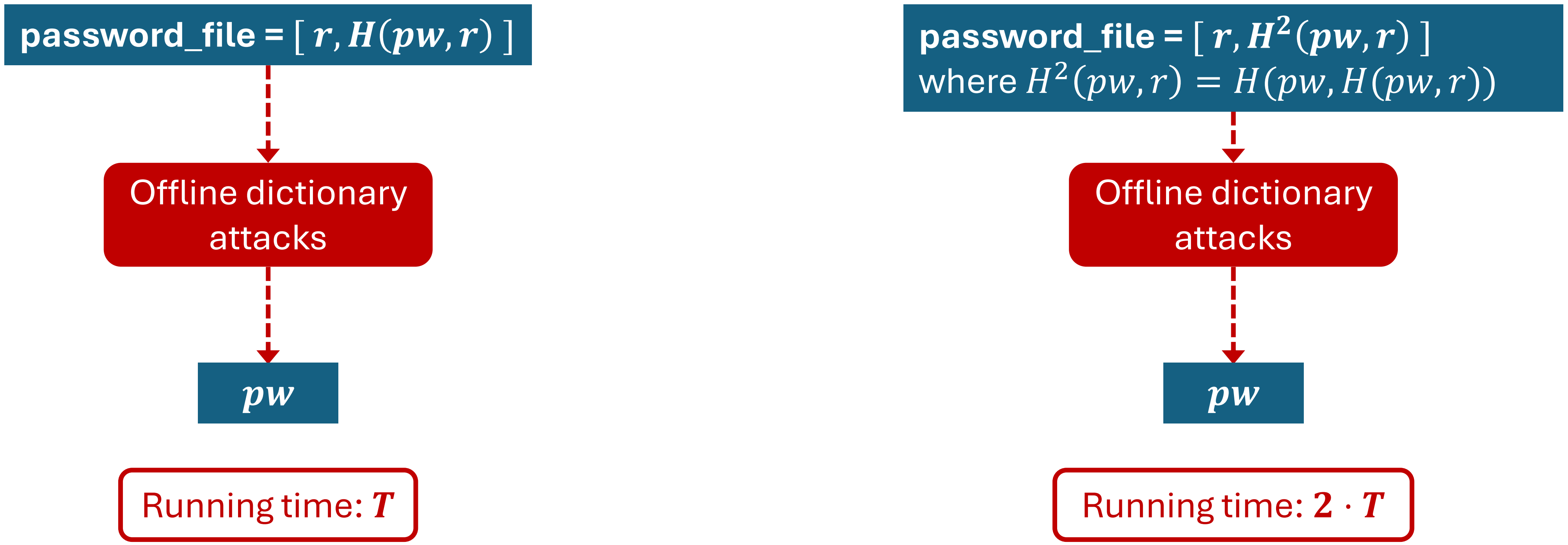
$\boldsymbol{pw}$

Running time: $\boldsymbol{T}$

password_file = $[\, \boldsymbol{r}, \boldsymbol{H^2}(\boldsymbol{pw}, \boldsymbol{r}) \,]$
where $H^2(pw, r) = H(pw, H(pw, r))$

The SCRAM protocol

- Add iteration in computing salted & hashed password:



password_file = $[\, \boldsymbol{r}, \boldsymbol{H}(\boldsymbol{pw}, \boldsymbol{r}) \,]$

Offline dictionary attacks

$\boldsymbol{pw}$

Running time: $\boldsymbol{T}$

password_file = $[\, \boldsymbol{r}, \boldsymbol{H}^{2}(\boldsymbol{pw}, \boldsymbol{r}) \,]$
where $H^{2}(pw, r) = H(pw, H(pw, r))$

Offline dictionary attacks

$\boldsymbol{pw}$

Running time: $\boldsymbol{2 \cdot T}$

The SCRAM protocol

- Add iteration in computing salted & hashed password:

```
Iterate_hash_with_salt( password,  salt, num_of_iteration):
    // salt can be 16- or 32-byte
    // num_of_iteration can be 4096 or even 100,000
    // All variable are bytes with big-endian order

    $pw$ = password
    padded_salt = salt || b'\x00\x00\x00\x01' // Append a 4-byte string 0x00000001 (in hex)

    $hash_1 = \mathrm{HMAC}(pw, padded\_salt)$ // We use keyed HMAC, where the key to HMAC is the password
    For $i$ from 2 to num_of_iteration:     // Iteratively evaluate the HMAC of pw and previous HMAC
        $hash_i = \mathrm{HMAC}(pw, hash_{i-1})$

    Password_file = $hash_1 \oplus hash_2 \oplus \cdots \oplus hash_{\mathrm{num\_of\_iteration}}$ // One integrate this part into the loop
    return Password_file
```

The SCRAM protocol

- Add iteration in computing salted & hashed password:

A simpler description:
(using the notation $\boldsymbol{H^n(pw, r)}$ = Iterate_hash_with_salt($\boldsymbol{pw, r, n}$)

Given $\boldsymbol{r, n, pw}$:

$\mathrm{U}_1 = \mathbf{HMAC}(\boldsymbol{pw}, \boldsymbol{r}$ || b'\x00\x00\x00\x01’)

$$\mathrm{U}_2 = \mathbf{HMAC}(\boldsymbol{pw}, \mathrm{U}_1)$$

$$\vdots$$

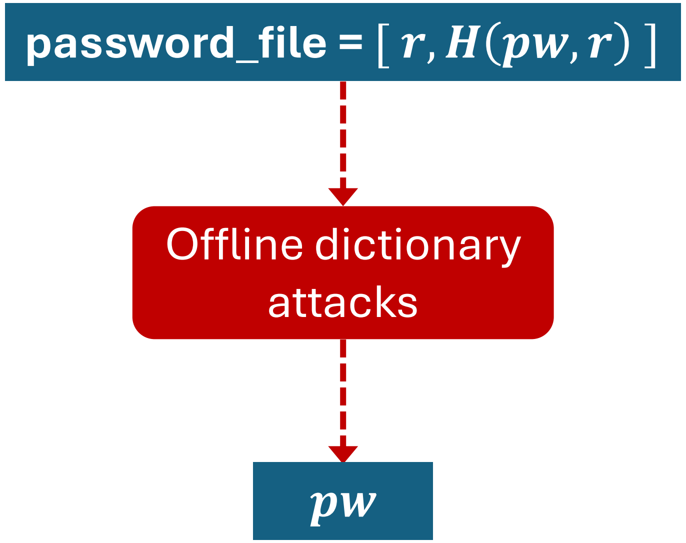
$$\mathrm{U}_{n-1} = \mathbf{HMAC}(\boldsymbol{pw}, \mathrm{U}_{n-2})$$

$$\mathrm{U}_n = \mathbf{HMAC}(\boldsymbol{pw}, \mathrm{U}_{n-1})$$

We compute $\boldsymbol{H^n(pw, r)} = \mathrm{U}_1 \oplus \mathrm{U}_2 \oplus \cdots \oplus \mathrm{U}_{\mathrm{n}-1} \oplus \mathrm{U}_{\mathrm{n}}$

The SCRAM protocol

- Add iteration in computing salted & hashed password:



password_file = [$r, H(pw, r)$]

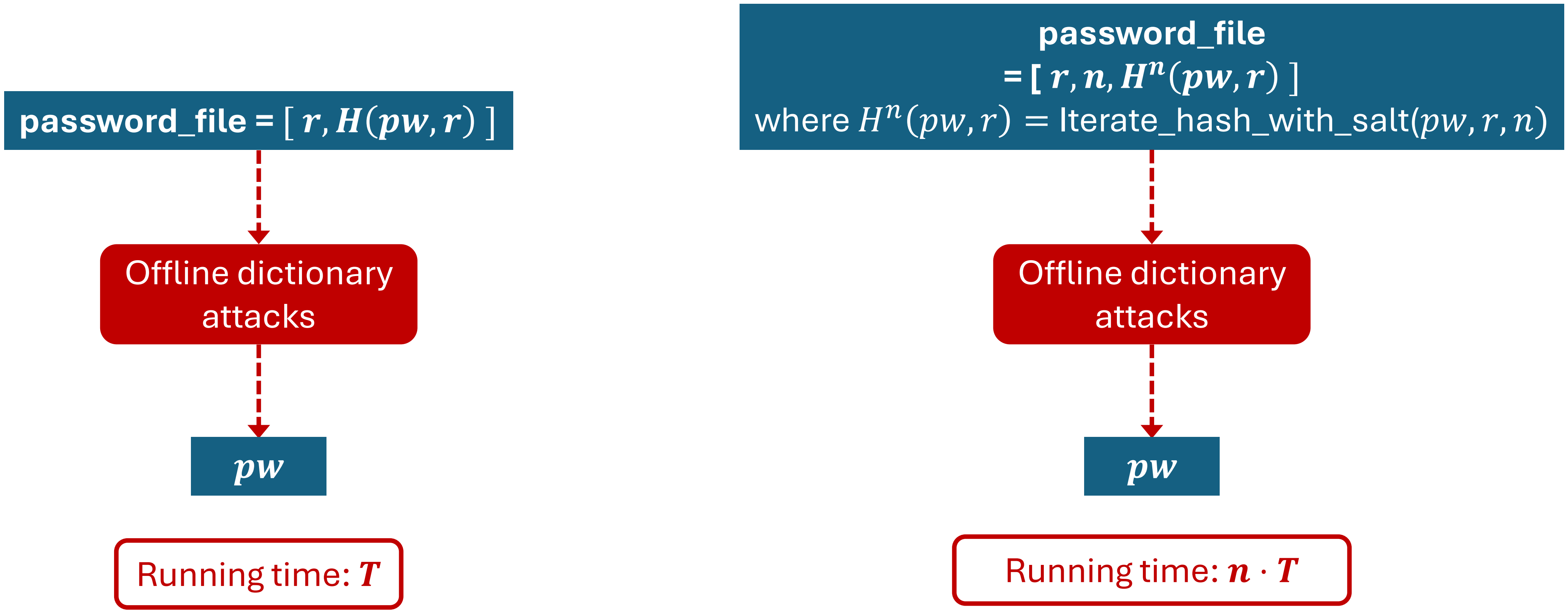
Offline dictionary attacks

pw

Running time: T

The SCRAM protocol

- Add iteration in computing salted & hashed password:



password_file = $[\, \boldsymbol{r}, \boldsymbol{H}(\boldsymbol{pw}, \boldsymbol{r}) \,]$

Offline dictionary attacks

$\boldsymbol{pw}$

Running time: $\boldsymbol{T}$

password_file
= $[\, \boldsymbol{r}, \boldsymbol{n}, \boldsymbol{H}^{\boldsymbol{n}}(\boldsymbol{pw}, \boldsymbol{r}) \,]$
where $H^n(pw, r)$ = Iterate_hash_with_salt(pw, r, n)

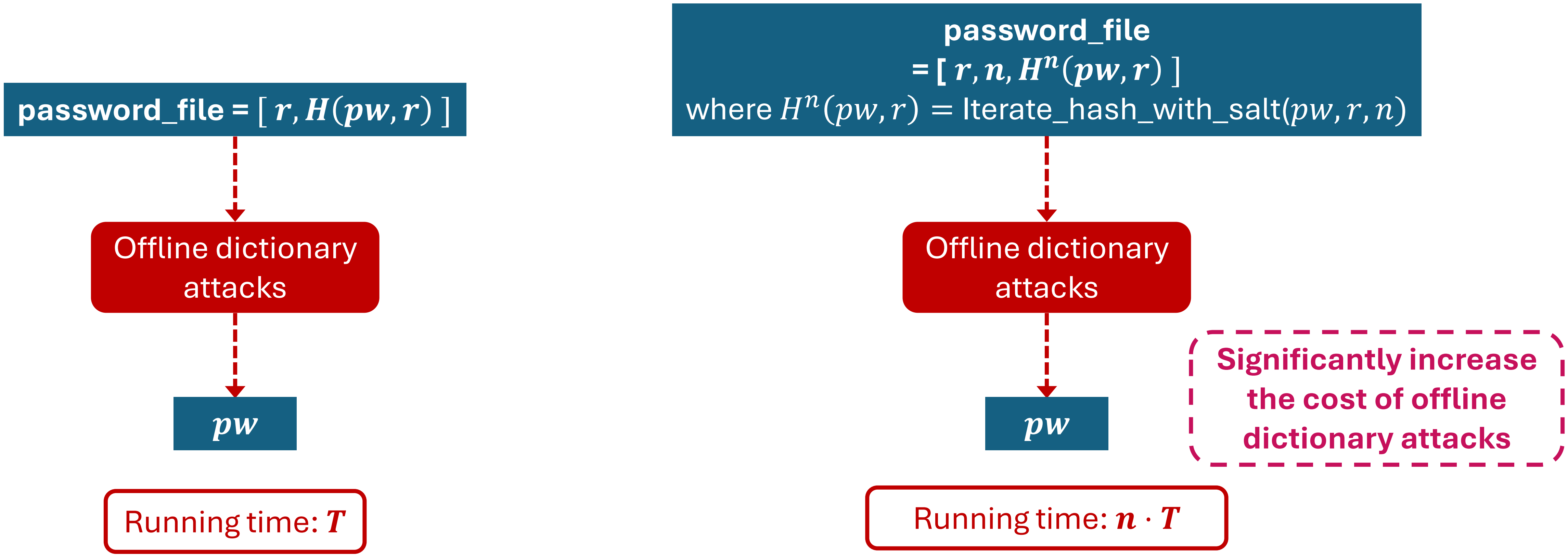
Offline dictionary attacks

$\boldsymbol{pw}$

Running time: $\boldsymbol{n} \cdot \boldsymbol{T}$

The SCRAM protocol

- Add iteration in computing salted & hashed password:



password_file = [$\boldsymbol{r, H(pw, r)}$]

Offline dictionary attacks

$\boldsymbol{pw}$

Running time: $\boldsymbol{T}$

password_file
= [$\boldsymbol{r, n, H^n(pw, r)}$]
where $H^n(pw, r)$ = Iterate_hash_with_salt(pw, r, n)

Offline dictionary attacks

$\boldsymbol{pw}$

Significantly increase the cost of offline dictionary attacks

Running time: $\boldsymbol{n \cdot T}$

The SCRAM protocol

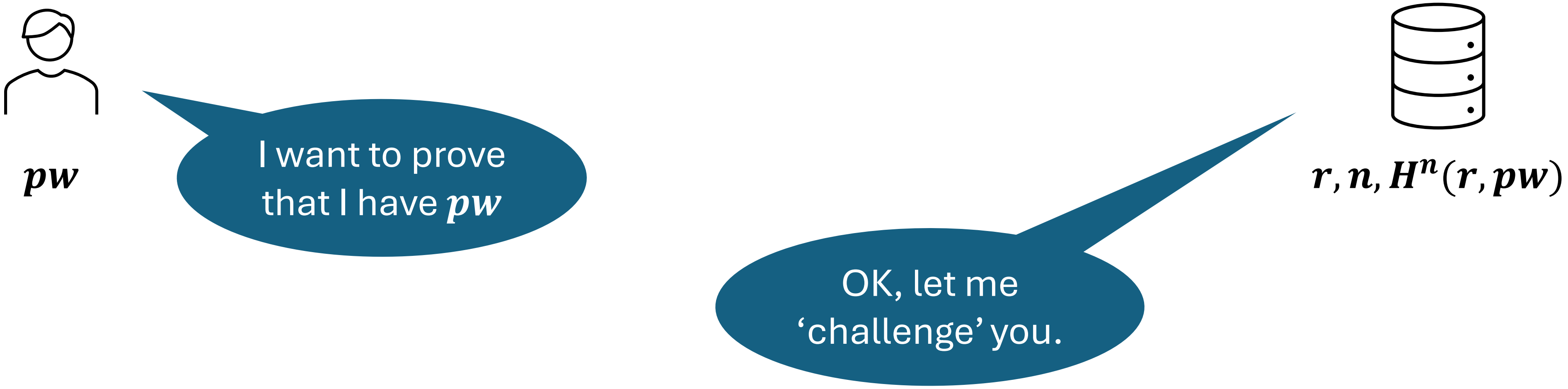
- Challenge-response paradigm

pw

$r, n, H^n(r, pw)$

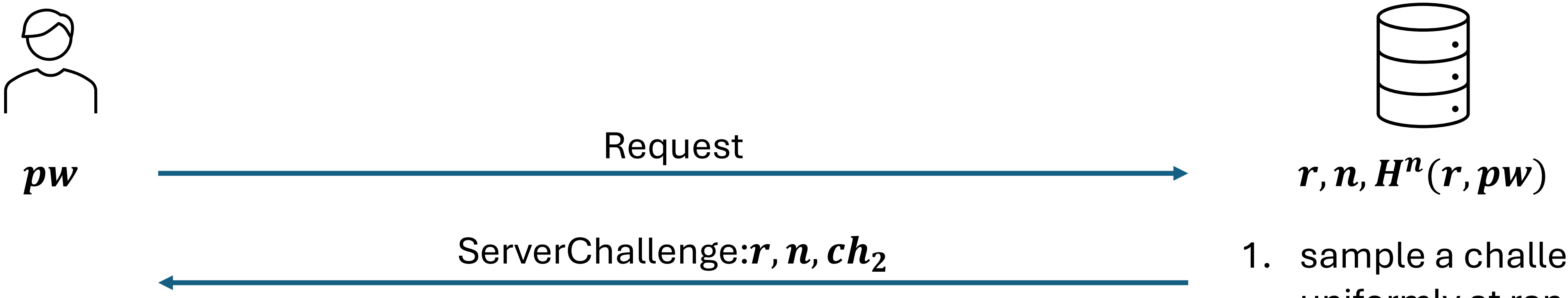
The SCRAM protocol

- Challenge-response paradigm



The SCRAM protocol

- Challenge-response paradigm



pw

Request

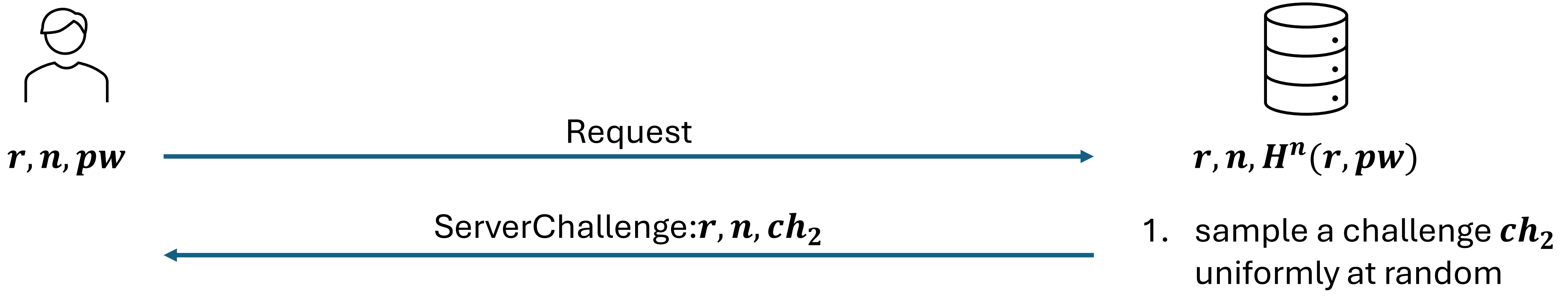
$r, n, H^n(r, pw)$

ServerChallenge:r, n, ch_2

1. sample a challenge ch_2 uniformly at random

The SCRAM protocol

- Challenge-response paradigm



$\boldsymbol{r, n, pw}$

Request

$\boldsymbol{r, n, H^n(r, pw)}$

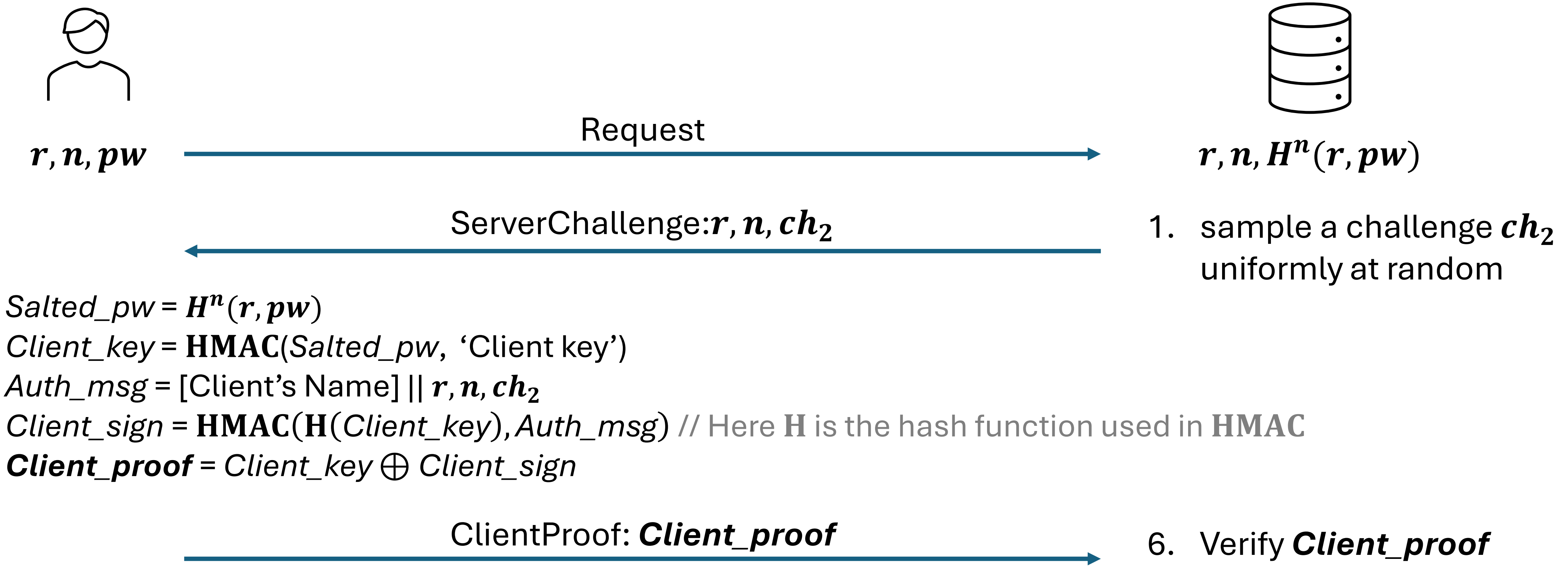
ServerChallenge:$\boldsymbol{r, n, ch_2}$

1. sample a challenge $\boldsymbol{ch_2}$ uniformly at random

2. *Salted_pw* = $\boldsymbol{H^n}(\boldsymbol{r}, \boldsymbol{pw})$
3. *Client_key* = $\mathbf{HMAC}$(*Salted_pw*, ‘Client key’)
4. *Auth_msg* = [Client’s Name] || $\boldsymbol{r, n, ch_2}$
5. *Client_sign* = $\mathbf{HMAC}(\mathbf{H}($*Client_key*$),$ *Auth_msg*$)$ // Here $\mathbf{H}$ is the hash function used in $\mathbf{HMAC}$
6. ***Client_proof*** = *Client_key* $\oplus$ *Client_sign*

The SCRAM protocol

- Challenge-response paradigm



$\boldsymbol{r, n, pw}$

Request

$\boldsymbol{r, n, H^n(r, pw)}$

ServerChallenge:$\boldsymbol{r, n, ch_2}$

1. sample a challenge $\boldsymbol{ch_2}$ uniformly at random

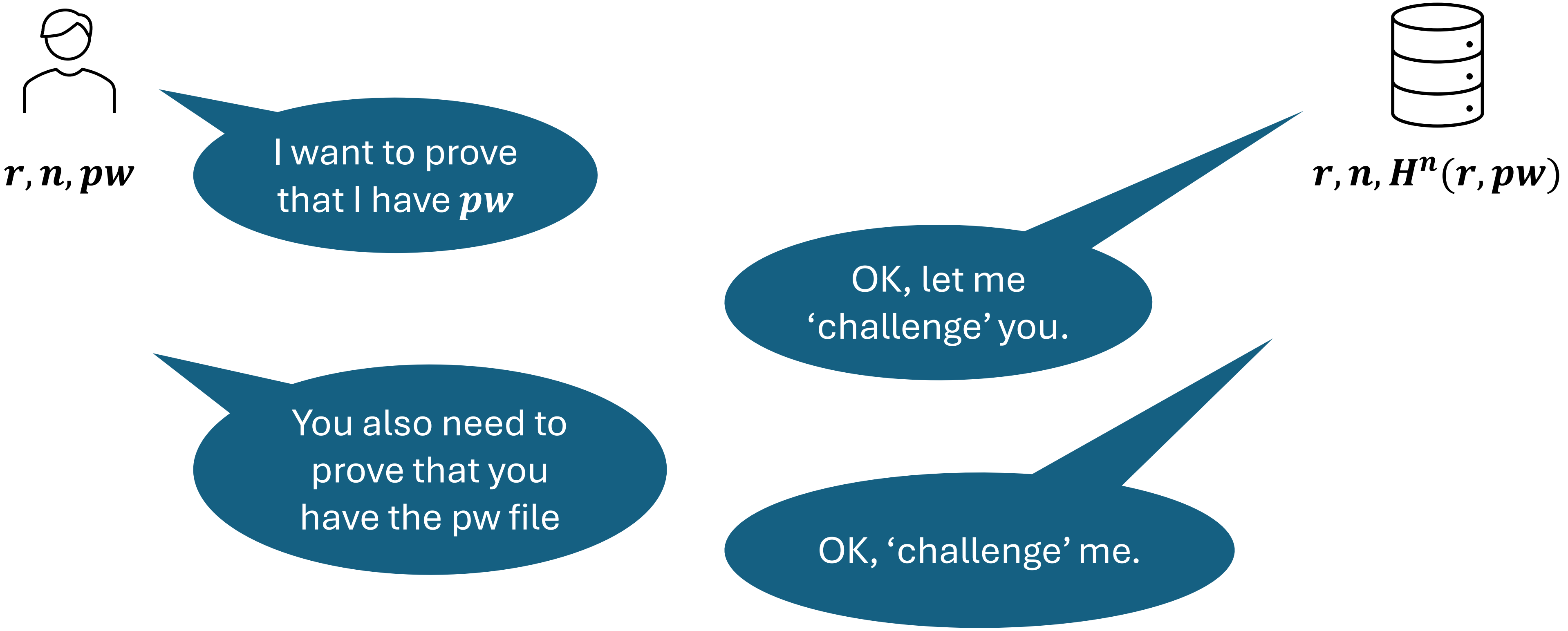
2. *Salted_pw* = $\boldsymbol{H^n(r, pw)}$
3. *Client_key* = $\mathbf{HMAC}$(*Salted_pw*, ‘Client key’)
4. *Auth_msg* = [Client’s Name] || $\boldsymbol{r, n, ch_2}$
5. *Client_sign* = $\mathbf{HMAC}(\mathbf{H}($*Client_key*$),$ *Auth_msg*$)$ // Here $\mathbf{H}$ is the hash function used in $\mathbf{HMAC}$
6. ***Client_proof*** = *Client_key* $\oplus$ *Client_sign*

ClientProof: ***Client_proof***

6. Verify ***Client_proof***

The SCRAM protocol

- Challenge-response paradigm



$\boldsymbol{r, n, pw}$

I want to prove that I have $\boldsymbol{pw}$

$\boldsymbol{r, n, H^n(r, pw)}$

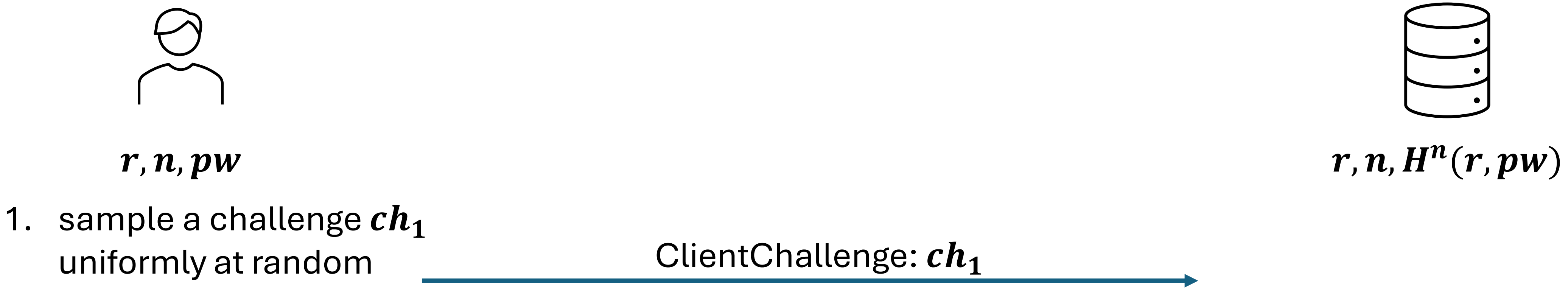
OK, let me 'challenge' you.

You also need to prove that you have the pw file

OK, 'challenge' me.

The SCRAM protocol

- Challenge-response paradigm



$\boldsymbol{r, n, pw}$

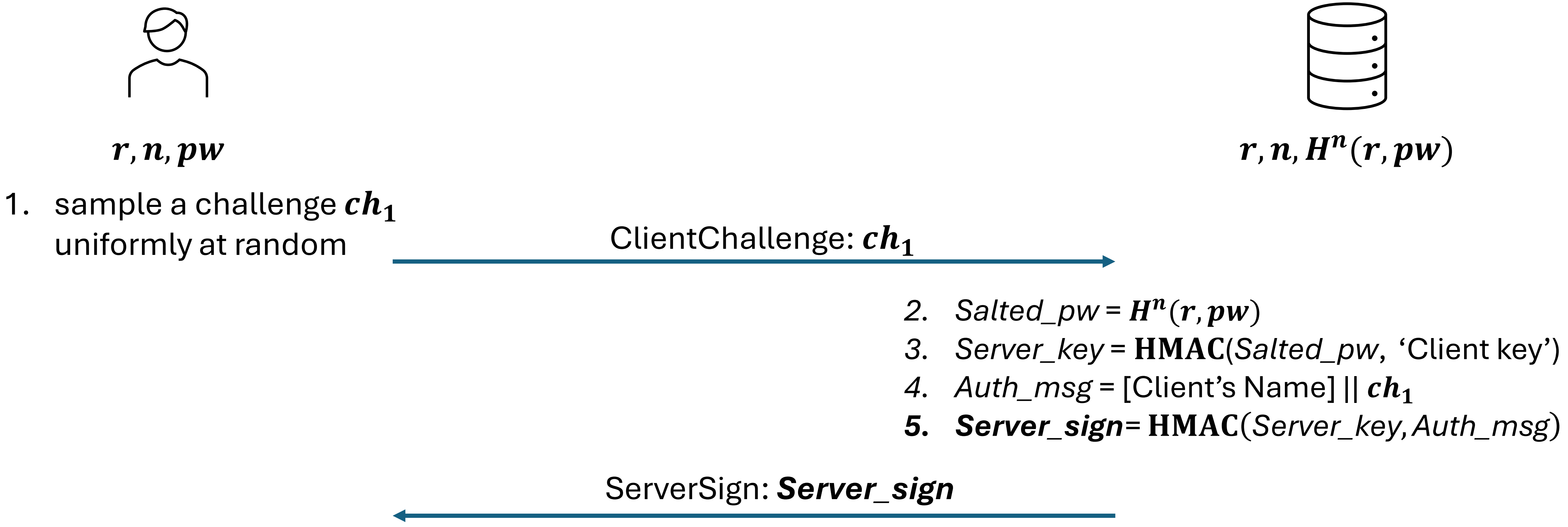
$\boldsymbol{r, n, H^n(r, pw)}$

1. sample a challenge $\boldsymbol{ch_1}$ uniformly at random

ClientChallenge: $\boldsymbol{ch_1}$

The SCRAM protocol

- Challenge-response paradigm



$\boldsymbol{r, n, pw}$

$\boldsymbol{r, n, H^n(r, pw)}$

1. sample a challenge $\boldsymbol{ch_1}$ uniformly at random

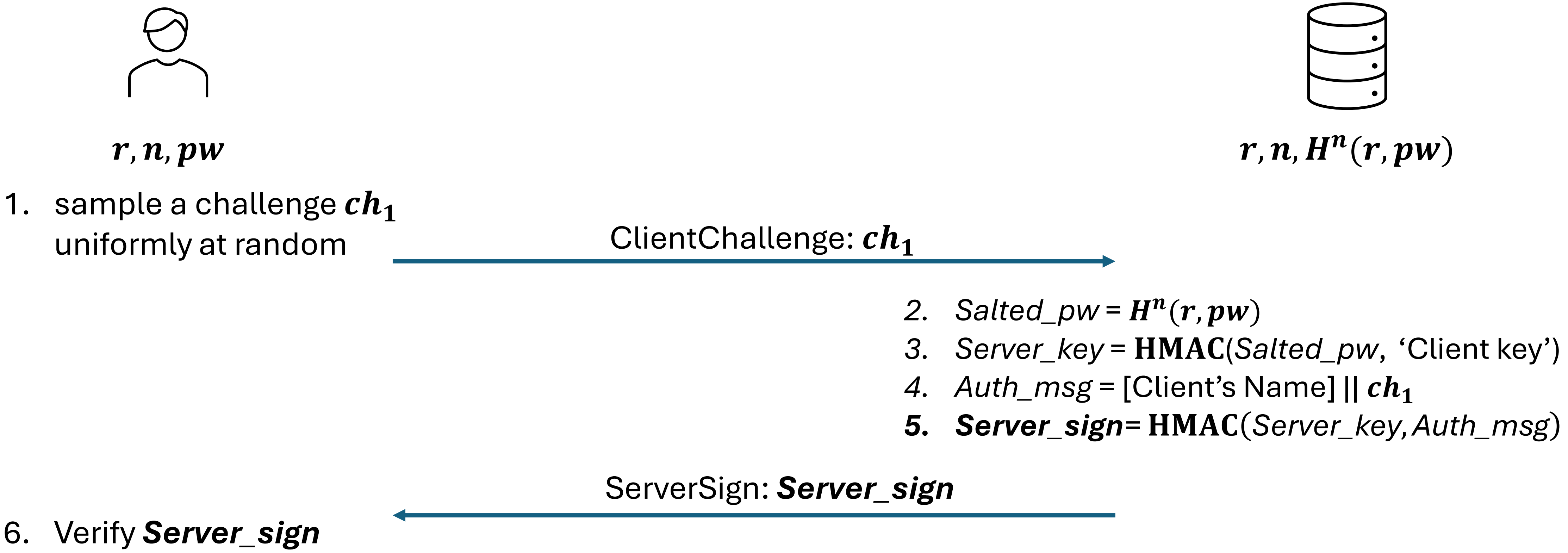
ClientChallenge: $\boldsymbol{ch_1}$

2. *Salted_pw* = $\boldsymbol{H^n(r, pw)}$
3. *Server_key* = $\mathbf{HMAC}$(*Salted_pw*, ‘Client key’)
4. *Auth_msg* = [Client’s Name] || $\boldsymbol{ch_1}$
5. ***Server_sign***= $\mathbf{HMAC}$(*Server_key*, *Auth_msg*)

ServerSign: ***Server_sign***

The SCRAM protocol

- Challenge-response paradigm



$\boldsymbol{r, n, pw}$

1. sample a challenge $\boldsymbol{ch_1}$ uniformly at random

ClientChallenge: $\boldsymbol{ch_1}$

$\boldsymbol{r, n, H^n(r, pw)}$

2. *Salted_pw* = $\boldsymbol{H^n(r, pw)}$
3. *Server_key* = $\mathbf{HMAC}$(*Salted_pw*, ‘Client key’)
4. *Auth_msg* = [Client’s Name] || $\boldsymbol{ch_1}$
5. ***Server_sign***= $\mathbf{HMAC}$(*Server_key*, *Auth_msg*)

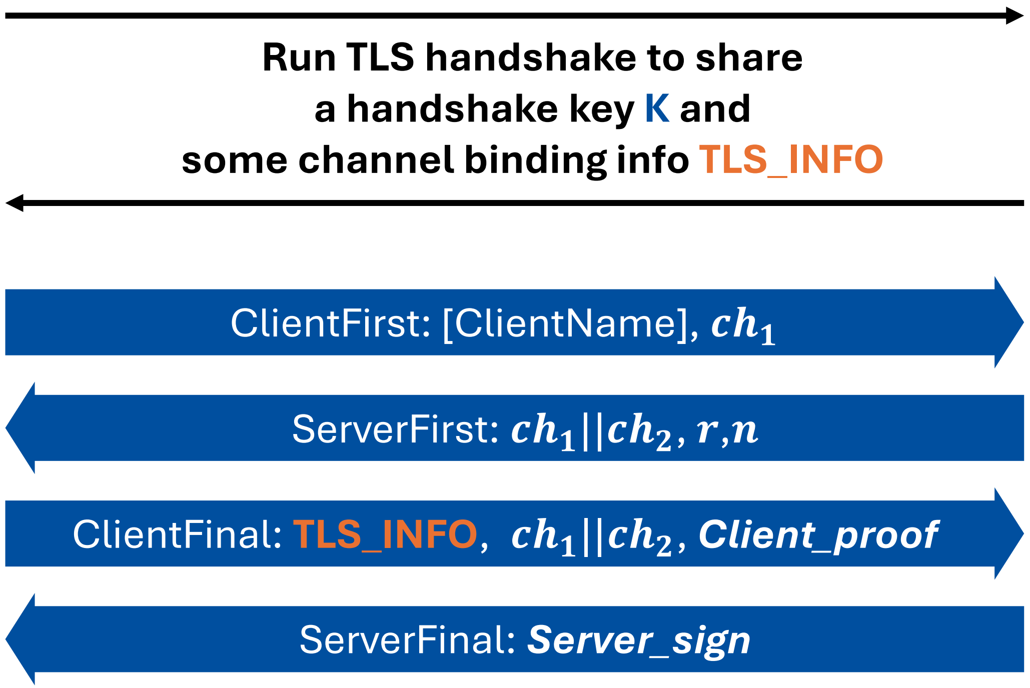
ServerSign: ***Server_sign***

6. Verify ***Server_sign***

The SCRAM protocol

Account = [ClientName]
password = $\boldsymbol{pw}$
where pw is some string

User	password_file
Runzhi	$\boldsymbol{r, n, H^n(r, pw)}$
Tom	$r_2, n, H^n(r_2, pw_2)$
…	…



Run TLS handshake to share a handshake key K and some channel binding info TLS_INFO

1. Pick a random client challenge”$\boldsymbol{ch_1}$

ClientFirst: [ClientName], $\boldsymbol{ch_1}$

2. Pick a random server challenge”ch_2

ServerFirst: $\boldsymbol{ch_1||ch_2, r, n}$

3. Compute ***Client_proof*** using **Auth_msg**

ClientFinal: **TLS_INFO**, $\boldsymbol{ch_1||ch_2}$, ***Client_proof***

4. Verify ***Client_proof.*** If valid: Compute ***Server_sign*** using **Auth_msg**

ServerFinal: ***Server_sign***

5. Verify ***Server_sign***

Auth_msg = [ClientName] || $\boldsymbol{ch_1||ch_2}$ || $\boldsymbol{r}$ || $\boldsymbol{n}$ || TLS_INFO

The SCRAM protocol

- Main idea:
  1. Add iteration in computing salted & hashed password
  2. Challenge-response Mechanism
  3. Run over TLS

- Used in some systems that require higher security guarantees…
  - ➢ IMAP / POP / SMTP / …
  - ➢ Database Authentication (e.g., MongoDB)…

Password-based Authenticated Key Exchange

- Previous protocols: TLS + salted & hashed (& iterated) passwords
  - Advantages: Simple, rely on known constructions

Password-based Authenticated Key Exchange

- Previous protocols: TLS + salted & hashed (& iterated) passwords
  - Advantages: Simple, rely on known constructions

- **Not entirely satisfactory:**
  - If the handshake protocol is not secure, then they can be trivially broken (leads to offline attacks immediately).

Password-based Authenticated Key Exchange

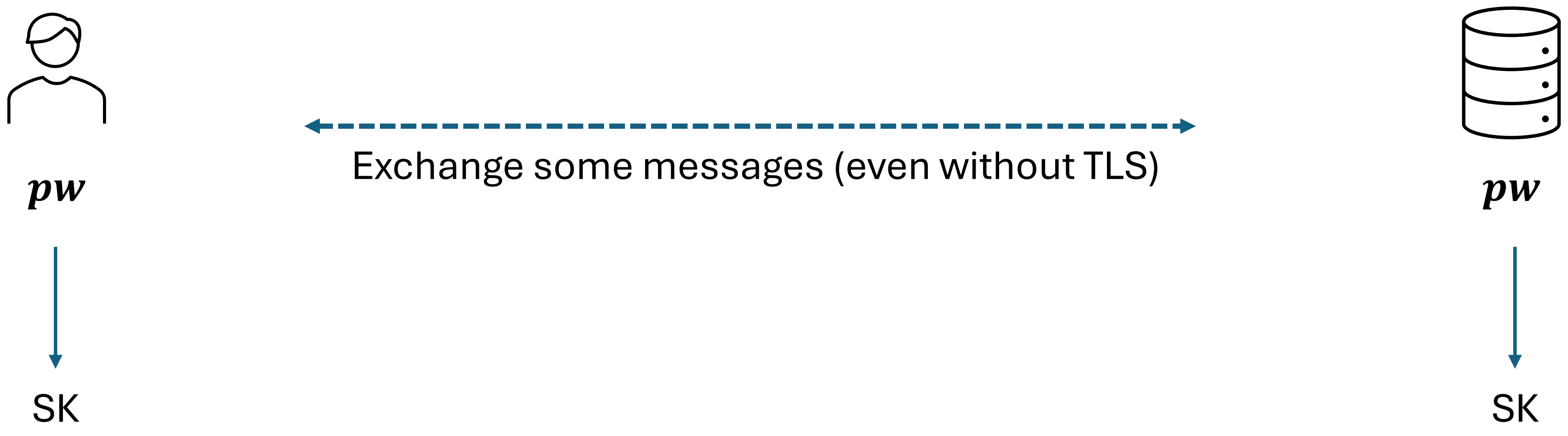
- Previous protocols: TLS + salted & hashed (& iterated) passwords
  - Advantages: Simple, rely on known constructions

- **Not entirely satisfactory:**
  - If the handshake protocol is not secure, then they can be trivially broken (leads to offline attacks immediately).

- **An alternative solution: Password-based Authenticated Key Exchange (PAKE)**

Password-based Authenticated Key Exchange

- **(Symmetric) PAKE:**



pw

Exchange some messages (even without TLS)

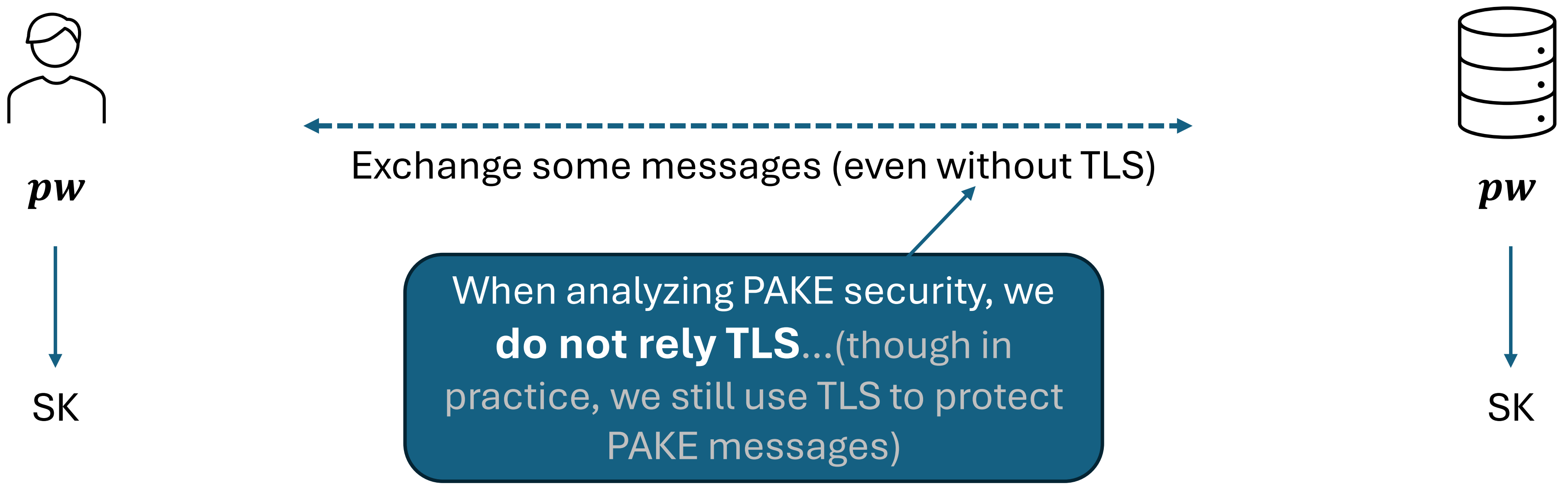
pw

SK

SK

Password-based Authenticated Key Exchange

- **(Symmetric) PAKE:**



pw

Exchange some messages (even without TLS)

pw

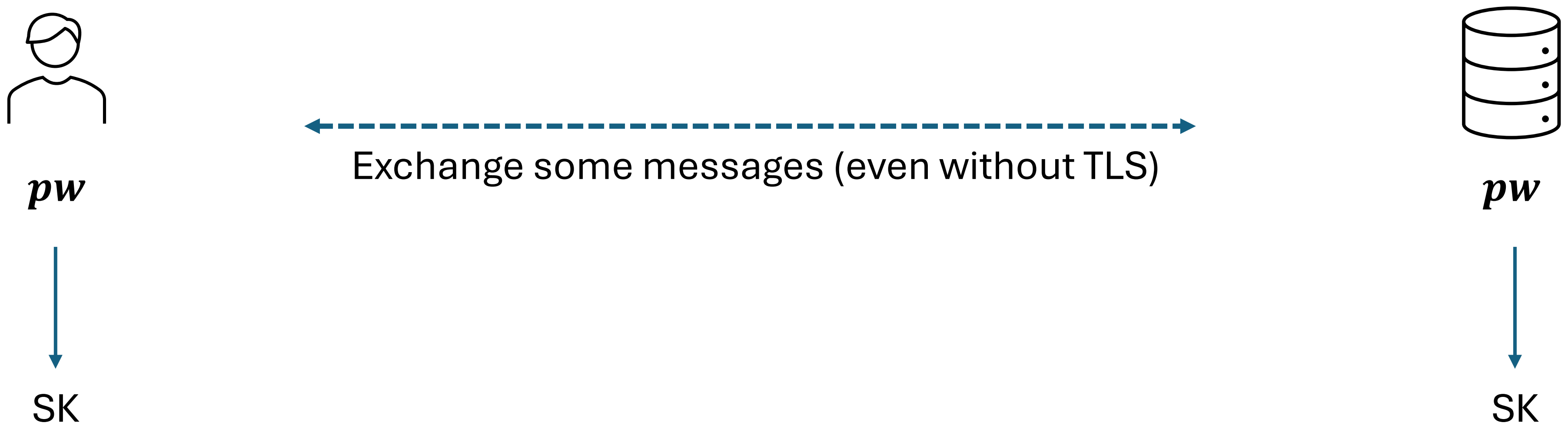
SK

When analyzing PAKE security, we **do not rely TLS**...(though in practice, we still use TLS to protect PAKE messages)

SK

Password-based Authenticated Key Exchange

- **(Symmetric) PAKE:**



pw

Exchange some messages (even without TLS)

pw

SK

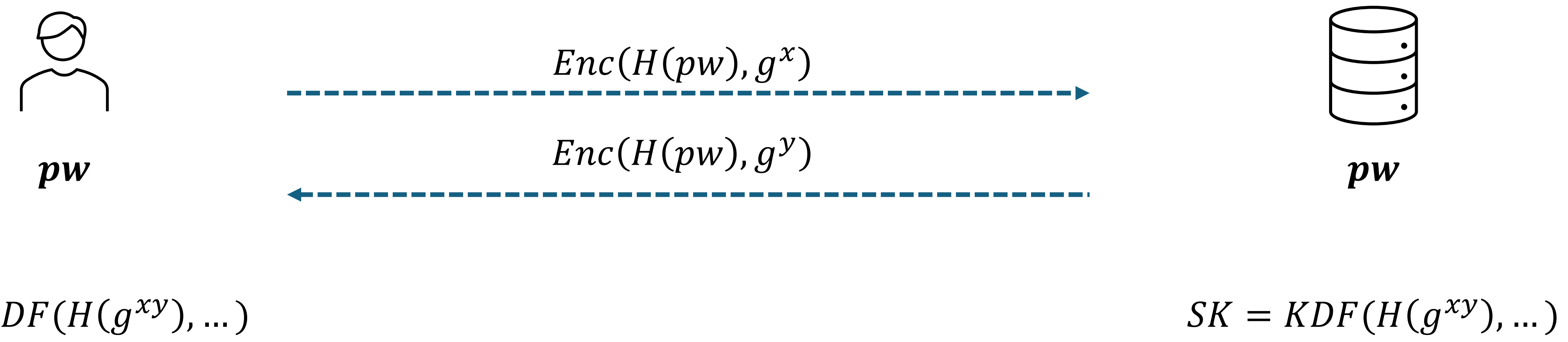
SK

Primary Goals:

- **(1) Resistance to Offline Dictionary attacks (2) The shared key SK is *pseudorandom***

Password-based Authenticated Key Exchange

- **Encrypted-Key-Exchange DH (EKE-DH) protocols:**
  - **Main idea:** Use pw to encrypt the underlying DH key exchange



$\boldsymbol{pw}$

$Enc(H(pw), g^x)$

$Enc(H(pw), g^y)$

$\boldsymbol{pw}$

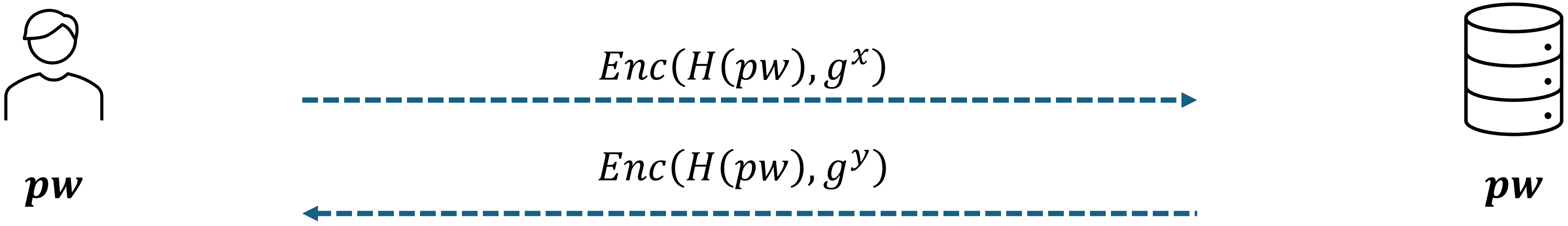
$SK = KDF(H(g^{xy}), \dots)$

$SK = KDF(H(g^{xy}), \dots)$

- Is it secure?

Password-based Authenticated Key Exchange

- **Encrypted-Key-Exchange DH (EKE-DH) protocols:**
  - **Main idea:** Use pw to encrypt the underlying DH key exchange



$Enc(H(pw), g^x)$

$Enc(H(pw), g^y)$

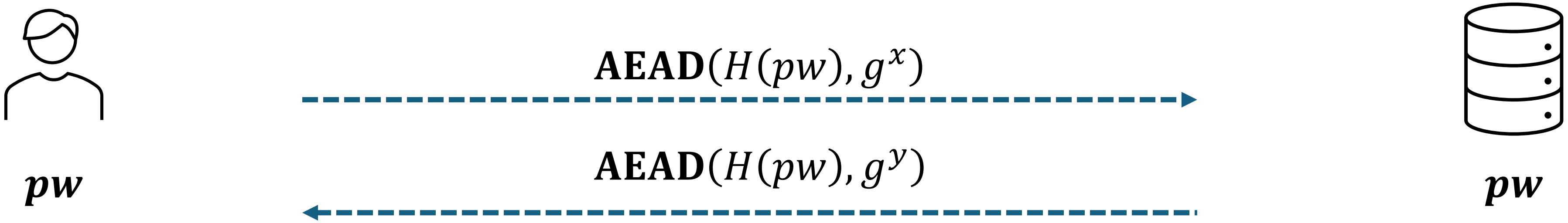
$\boldsymbol{pw}$ $\boldsymbol{pw}$

$SK = KDF(H(g^{xy}), \dots)$ $SK = KDF(H(g^{xy}), \dots)$

- Is it secure? **Depends on the encryption!**

Password-based Authenticated Key Exchange

- EKE-DH protocols **based on AEAD:**



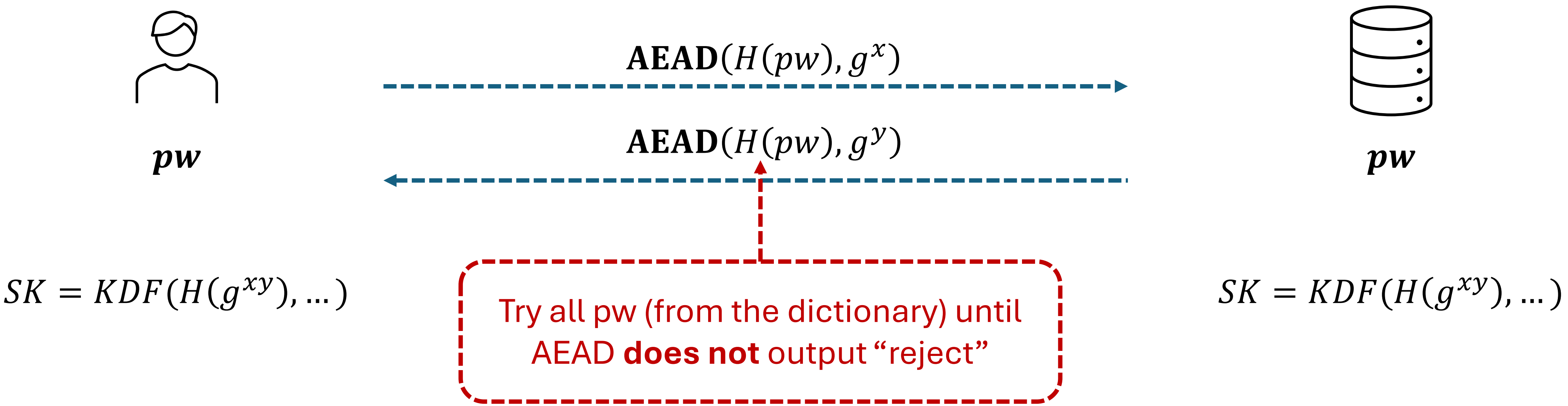
$SK = KDF(H(g^{xy}), \ldots)$

$SK = KDF(H(g^{xy}), \ldots)$

- Is it secure? (Hint: On invalid input key/ciphertext, AEAD may output “reject”)

Password-based Authenticated Key Exchange

- EKE-DH protocols **based on AEAD:**



$\textbf{pw}$

$\mathbf{AEAD}(H(pw), g^x)$

$\mathbf{AEAD}(H(pw), g^y)$

$\textbf{pw}$

$SK = KDF(H(g^{xy}), \ldots)$

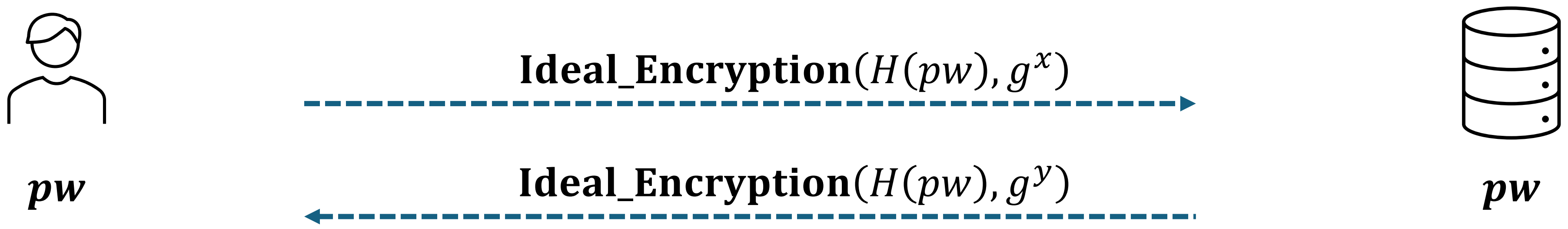
Try all pw (from the dictionary) until AEAD **does not** output "reject"

$SK = KDF(H(g^{xy}), \ldots)$

- Is it secure? (Hint: On invalid input key/ciphertext, AEAD may output "reject")

Password-based Authenticated Key Exchange

- EKE-DH protocols **based on an "ideal" encryption:**



$\textbf{Ideal\_Encryption}(H(pw), g^x)$

$\textbf{Ideal\_Encryption}(H(pw), g^y)$

$\boldsymbol{pw}$

$\boldsymbol{pw}$

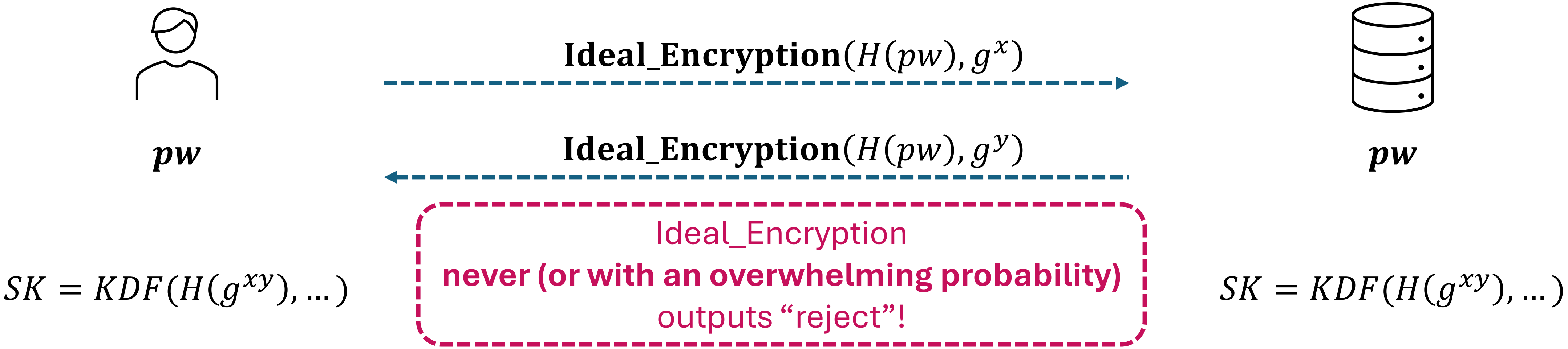
$SK = KDF(H(g^{xy}), \dots)$

$SK = KDF(H(g^{xy}), \dots)$

- The **ideal encryption** has the following properties:
    - Outputs of encryption and decryption **are (pseudo)random *even if the key has low entropy***
    - Namely, if the adversary does not have the correct pw, the outputs of encryption/decryption are some random group elements.

Password-based Authenticated Key Exchange

- EKE-DH protocols **based on an "ideal" encryption:**



$\boldsymbol{pw}$

$\textbf{Ideal\_Encryption}(H(pw), g^x)$

$\textbf{Ideal\_Encryption}(H(pw), g^y)$

$\boldsymbol{pw}$

$SK = KDF(H(g^{xy}), \ldots)$

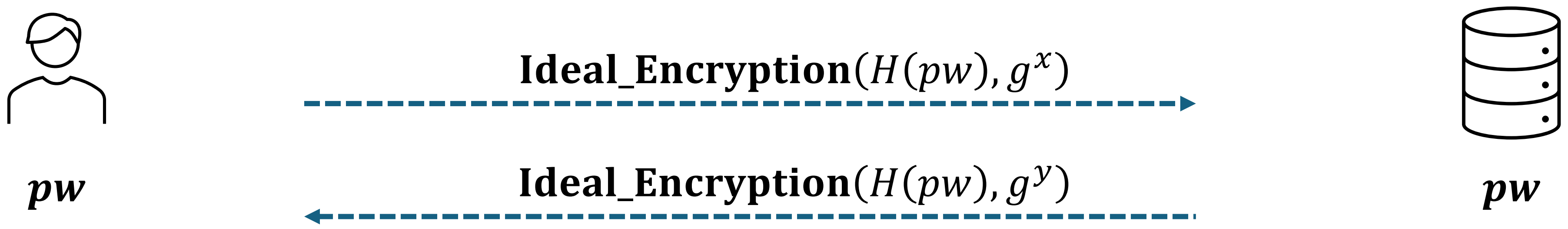
Ideal_Encryption
never (or with an overwhelming probability)
outputs "reject"!

$SK = KDF(H(g^{xy}), \ldots)$

- The **ideal encryption** has the following properties:
  - Outputs of encryption and decryption **are (pseudo)random *even if the key has low entropy***
  - Namely, if the adversary does not have the correct pw, the outputs of encryption/decryption are some random group elements.

Password-based Authenticated Key Exchange

- EKE-DH protocols **based on an “ideal” encryption:**



Ideal_Encryption$(H(pw), g^x)$
Ideal_Encryption$(H(pw), g^y)$
$\boldsymbol{pw}$
$\boldsymbol{pw}$

$SK = KDF(H(g^{xy}), \dots)$

$SK = KDF(H(g^{xy}), \dots)$

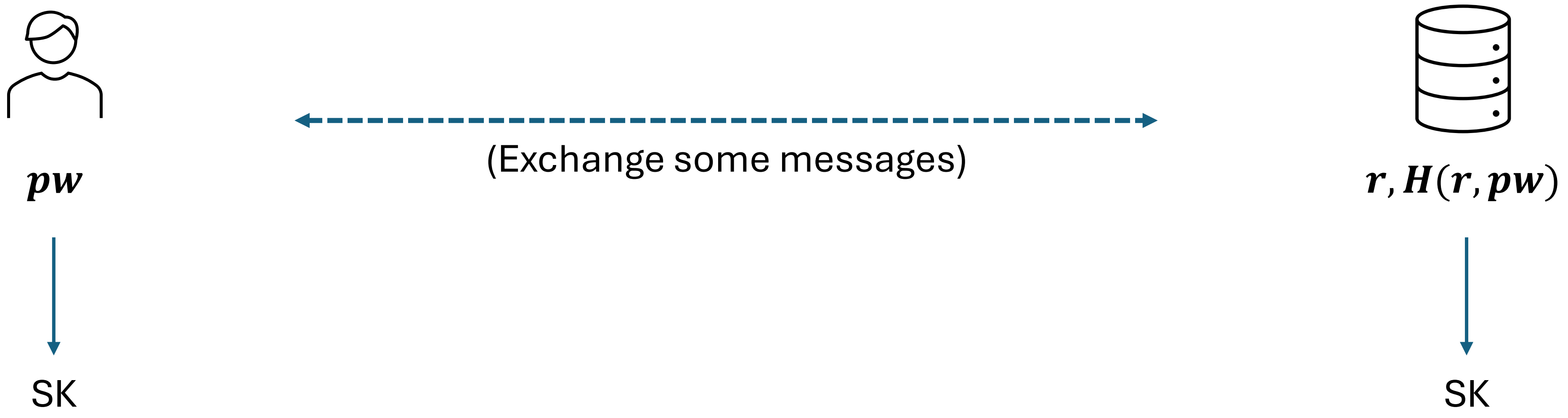
Ideal_Encryption
never (or with an overwhelming probability)
outputs “reject”!

- The **ideal encryption**
  - Outputs of encrypti ***has low entropy***
  - Namely, if the adver tion/decryption are some random group elements.

How to design it (with high efficiency & strong security): **Open problem**

Password-based Authenticated Key Exchange

- **Asymmetric PAKE (aPAKE):**



$\boldsymbol{pw}$

(Exchange some messages)

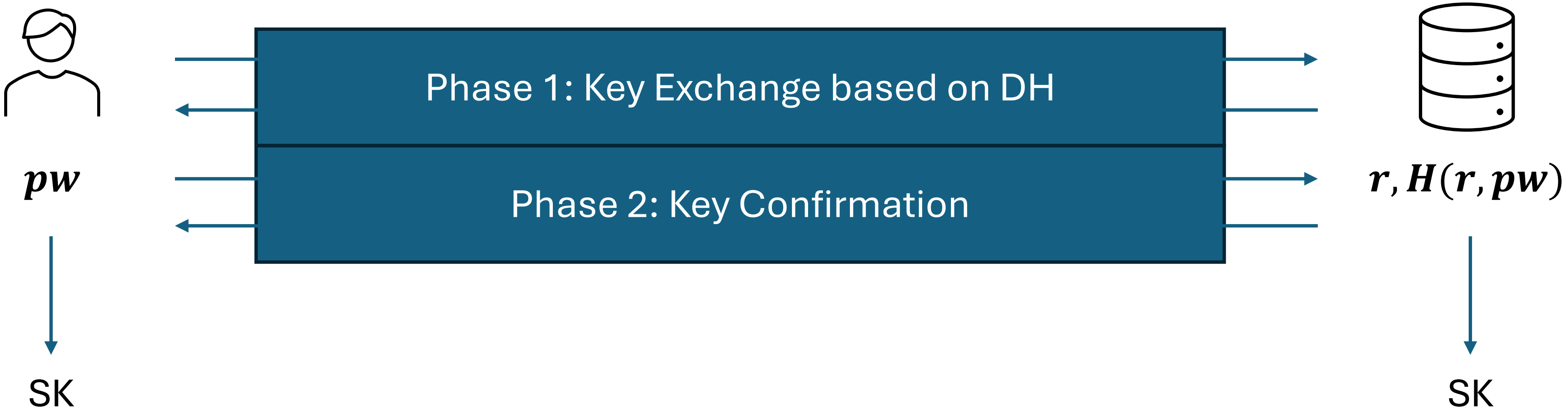
$\boldsymbol{r, H(r, pw)}$

SK

SK

Password-based Authenticated Key Exchange

- **Secure Remote Password (SRP) Protocol**



- Based on module integer groups / Not directly compatible with Elliptic Curves
- Apple ID Authentication / Blizzard Entertainment

Password-based Authenticated Key Exchange

- **Secure Remote Password (SRP) Protocol (version 6a)**

$\boldsymbol{pw}$

Public Parameters:

$\boldsymbol{q}$	a large prime
$\boldsymbol{N = 2q + 1}$	a prime (we call it as safe prime)
$\mathbb{G}$	a sub-group of $\mathbb{Z}_N$ with order $\boldsymbol{q}$
g	the generator of $\mathbb{G}$

Notations:

Let h be an integer in $\mathbb{Z}_N$.

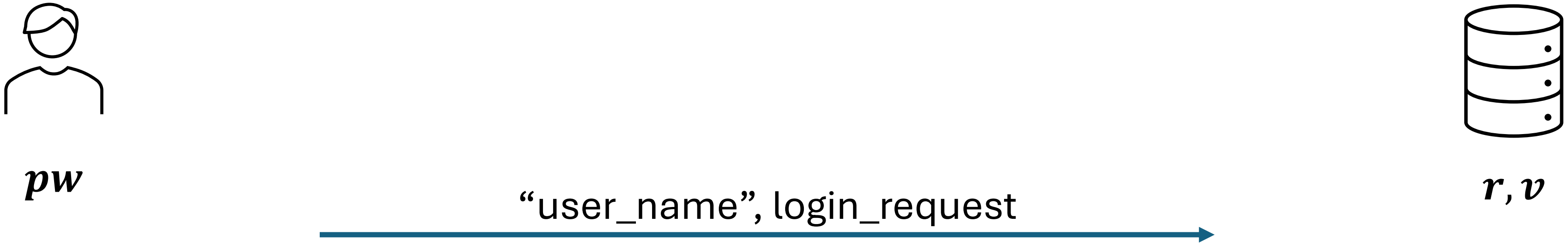
If $h \in \mathbb{G}$ and $x \in \mathbb{Z}_q$, then we denote $h^x := h^x \ mod\ N$

Password file:

$\boldsymbol{r, v = g^{H(r,\,[user\_name],\,pw)}}$

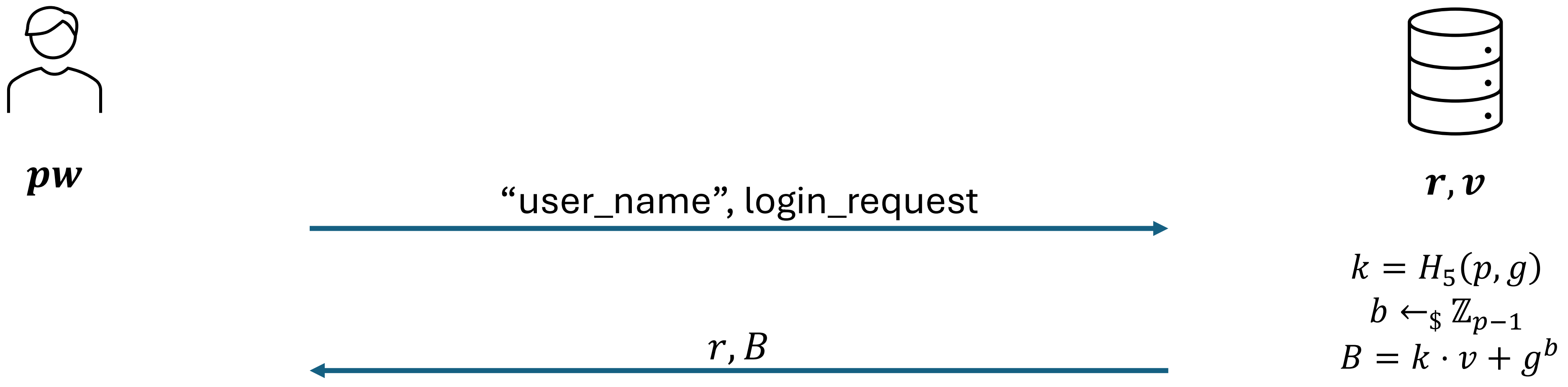
Password-based Authenticated Key Exchange

- **SRP-v6a: (1) Key Exchange phase** (2) Key Confirmation phase



Password-based Authenticated Key Exchange

- **SRP-v6a: (1) Key Exchange phase** (2) Key Confirmation phase



$\boldsymbol{pw}$

$\boldsymbol{r, v}$

“user_name”, login_request

$k = H_5(p, g)$

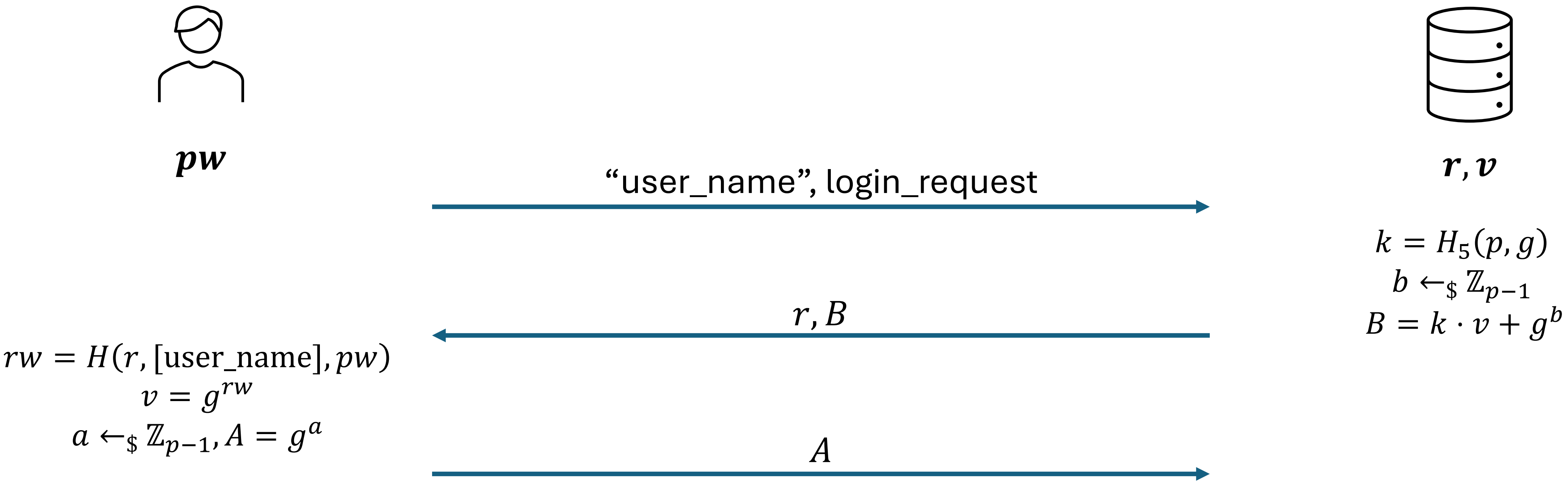
$b \leftarrow_{\$} \mathbb{Z}_{p-1}$

$B = k \cdot v + g^b$

r, B

Password-based Authenticated Key Exchange

- **SRP-v6a: (1) Key Exchange phase** (2) Key Confirmation phase



$\boldsymbol{pw}$

$\boldsymbol{r, v}$

“user_name”, login_request

$k = H_5(p, g)$

$b \leftarrow_\$ \mathbb{Z}_{p-1}$

$B = k \cdot v + g^b$

r, B

$rw = H(r, [\text{user\_name}], pw)$

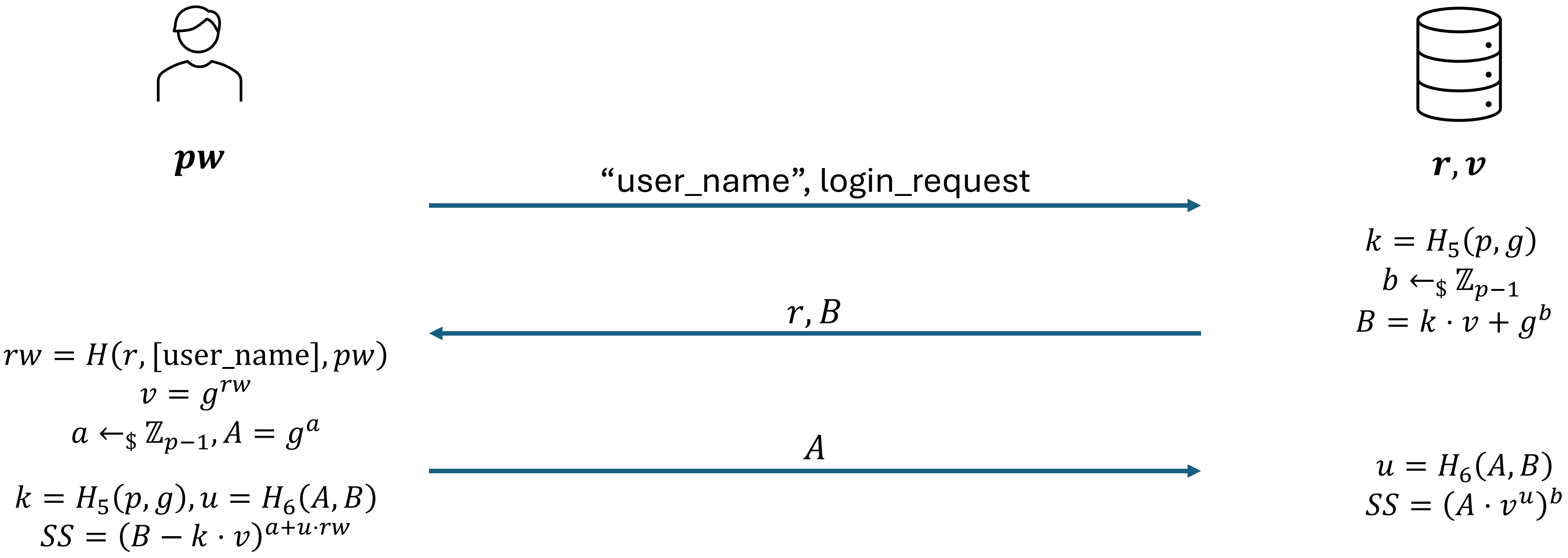
$v = g^{rw}$

$a \leftarrow_\$ \mathbb{Z}_{p-1}, A = g^a$

A

Password-based Authenticated Key Exchange

- **SRP-v6a: (1) Key Exchange phase** (2) Key Confirmation phase



$\boldsymbol{pw}$

$\boldsymbol{r, v}$

“user_name”, login_request

$k = H_5(p, g)$
$b \leftarrow_{\$} \mathbb{Z}_{p-1}$
$B = k \cdot v + g^b$

r, B

$rw = H(r, [\text{user\_name}], pw)$
$v = g^{rw}$
$a \leftarrow_{\$} \mathbb{Z}_{p-1}, A = g^a$

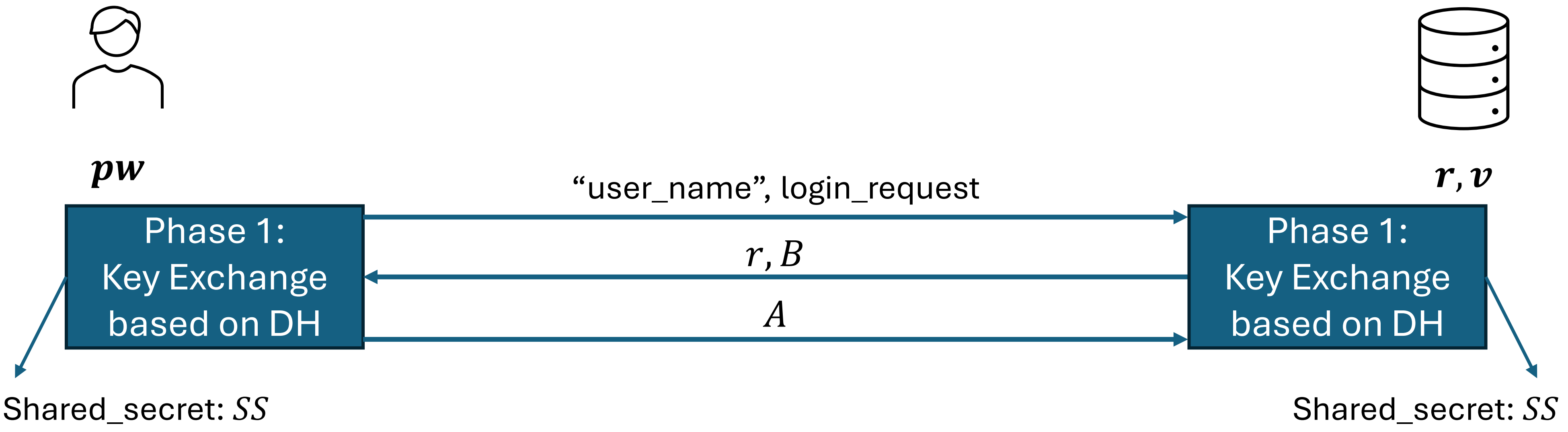
A

$k = H_5(p, g), u = H_6(A, B)$
$SS = (B - k \cdot v)^{a + u \cdot rw}$

$u = H_6(A, B)$
$SS = (A \cdot v^u)^b$

Password-based Authenticated Key Exchange

- **SRP-v6a:** (1) Key Exchange phase **(2) Key Confirmation phase**



pw

r, v

Phase 1:
Key Exchange
based on DH

“user_name”, login_request

r, B

A

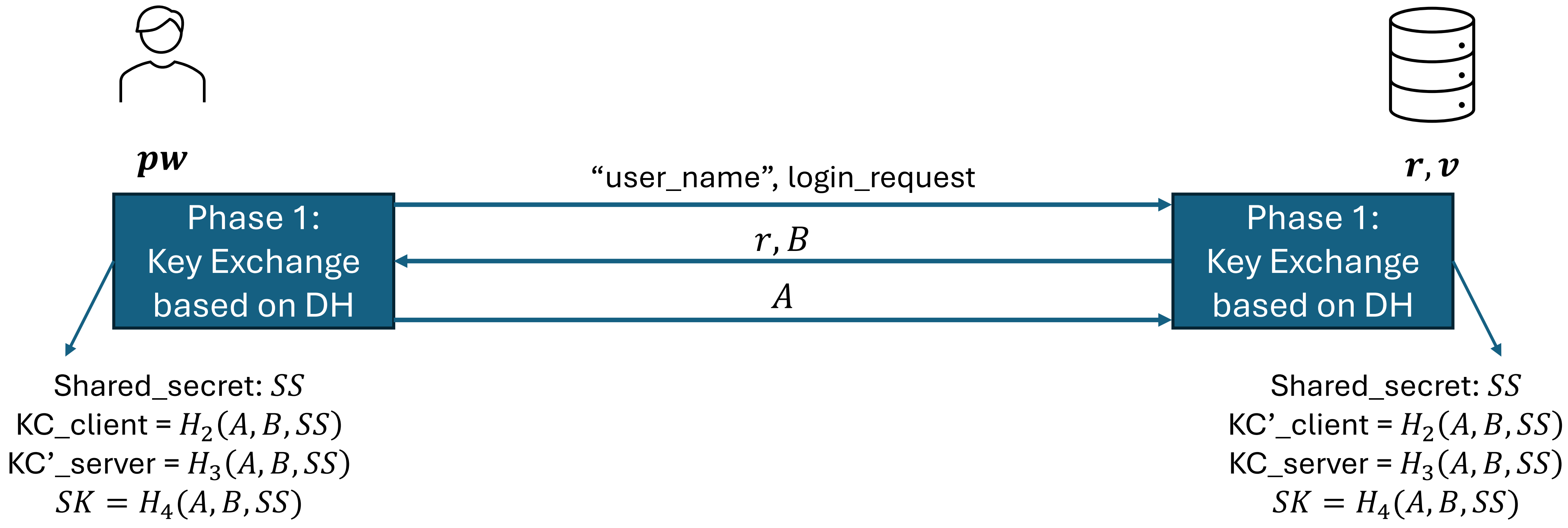
Phase 1:
Key Exchange
based on DH

Shared_secret: SS

Shared_secret: SS

Password-based Authenticated Key Exchange

- **SRP-v6a:** (1) Key Exchange phase **(2) Key Confirmation phase**



$\boldsymbol{pw}$

$\boldsymbol{r, v}$

Phase 1:
Key Exchange
based on DH

“user_name”, login_request

r, B

A

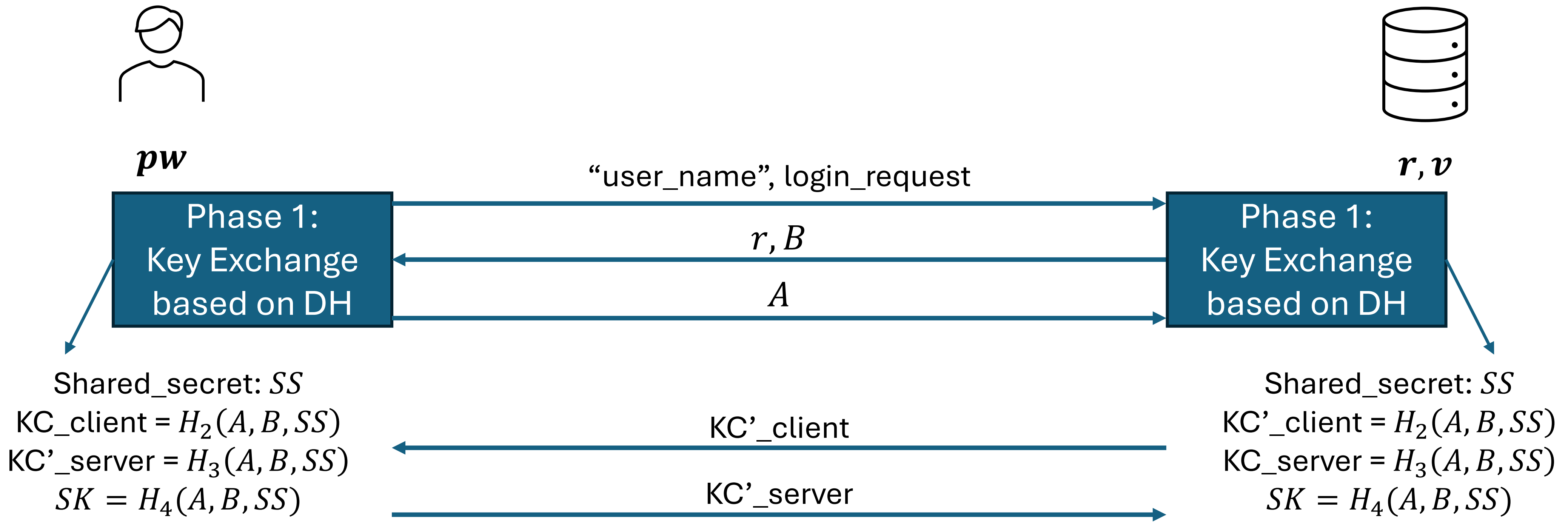
Phase 1:
Key Exchange
based on DH

Shared_secret: SS
KC_client = $H_2(A, B, SS)$
KC’_server = $H_3(A, B, SS)$
$SK = H_4(A, B, SS)$

Shared_secret: SS
KC’_client = $H_2(A, B, SS)$
KC_server = $H_3(A, B, SS)$
$SK = H_4(A, B, SS)$

Password-based Authenticated Key Exchange

- **SRP-v6a:** (1) Key Exchange phase **(2) Key Confirmation phase**



$\boldsymbol{pw}$

$\boldsymbol{r, v}$

Phase 1: Key Exchange based on DH

"user_name", login_request

r, B

A

Phase 1: Key Exchange based on DH

Shared_secret: SS
KC_client = $H_2(A, B, SS)$
KC'_server = $H_3(A, B, SS)$
$SK = H_4(A, B, SS)$

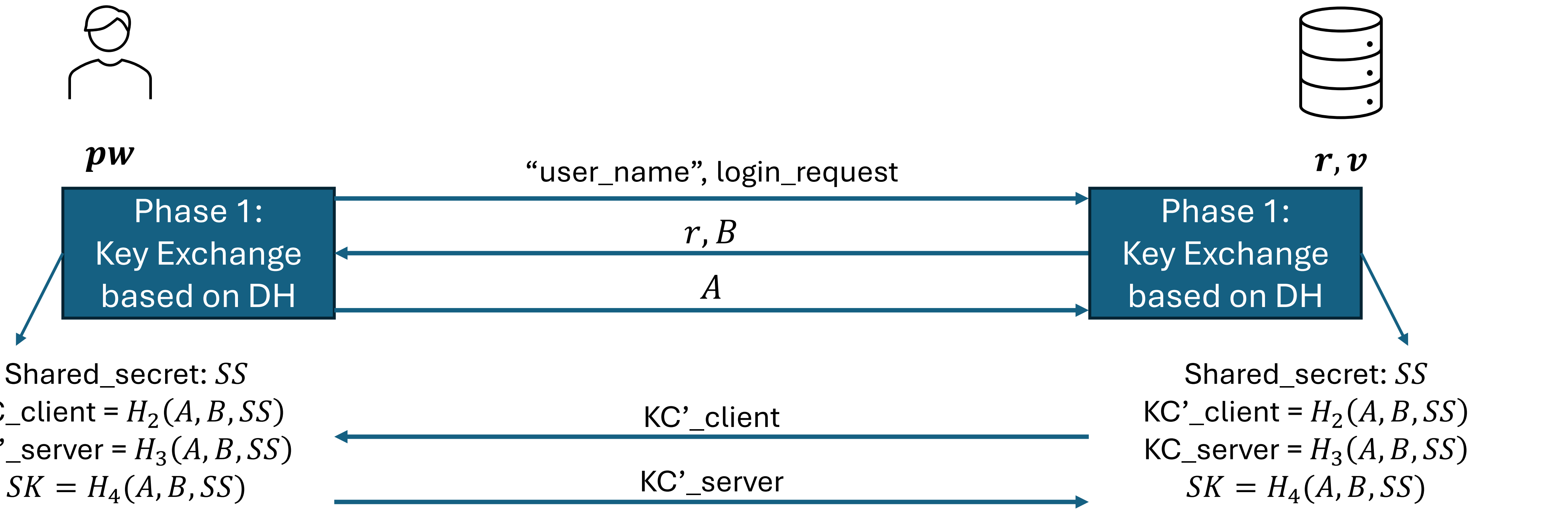
KC'_client

KC'_server

Shared_secret: SS
KC'_client = $H_2(A, B, SS)$
KC_server = $H_3(A, B, SS)$
$SK = H_4(A, B, SS)$

Password-based Authenticated Key Exchange

- **SRP-v6a:** (1) Key Exchange phase **(2) Key Confirmation phase**



pw

r, v

“user_name”, login_request

Phase 1: Key Exchange based on DH

r, B

A

Phase 1: Key Exchange based on DH

Shared_secret: SS
KC_client = $H_2(A, B, SS)$
KC’_server = $H_3(A, B, SS)$
$SK = H_4(A, B, SS)$

KC’_client

KC’_server

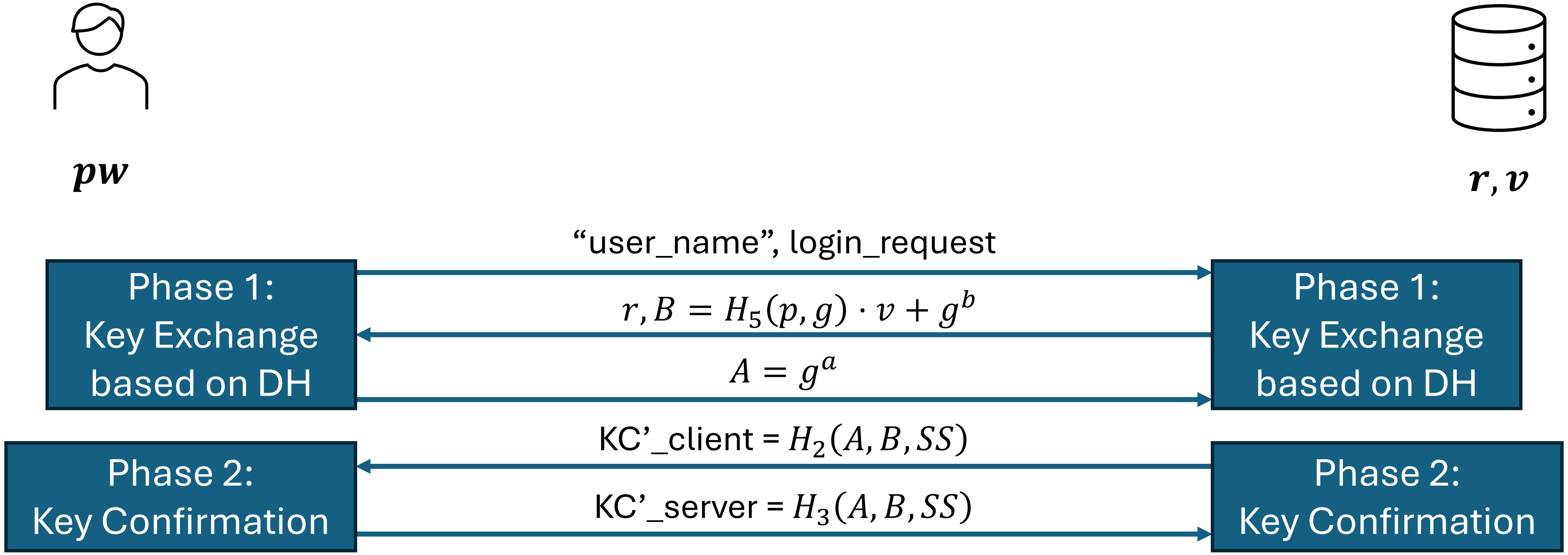
Shared_secret: SS
KC’_client = $H_2(A, B, SS)$
KC_server = $H_3(A, B, SS)$
$SK = H_4(A, B, SS)$

Accept SK iff KC’_client == KC_client

Accept SK iff KC’_server == KC_server

Password-based Authenticated Key Exchange

- **SRP-v6a:**



$\boldsymbol{pw}$

$\boldsymbol{r, v}$

Phase 1:
Key Exchange
based on DH

“user_name”, login_request

$r, B = H_5(p, g) \cdot v + g^b$

$A = g^a$

Phase 1:
Key Exchange
based on DH

Phase 2:
Key Confirmation

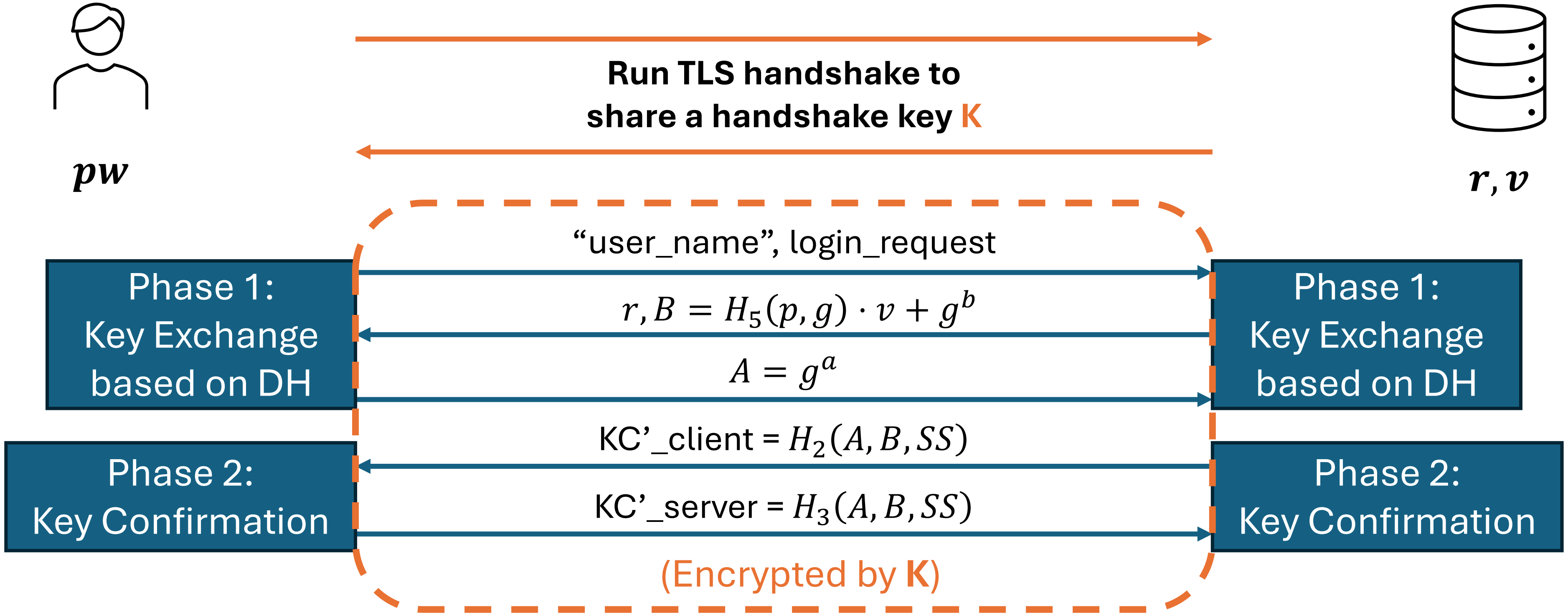
KC’_client = $H_2(A, B, SS)$

KC’_server = $H_3(A, B, SS)$

Phase 2:
Key Confirmation

Password-based Authenticated Key Exchange

- TLS + SRP-v6a:



pw
r, v
Run TLS handshake to share a handshake key K
Phase 1: Key Exchange based on DH
Phase 2: Key Confirmation
"user_name", login_request
$r, B = H_5(p, g) \cdot v + g^b$
$A = g^a$
KC'_client = $H_2(A, B, SS)$
KC'_server = $H_3(A, B, SS)$
(Encrypted by K)
Phase 1: Key Exchange based on DH
Phase 2: Key Confirmation

Homework

- Implement the SCRAM protocol (You do not need to use sockets, but your program should draw the message flows)
- **Bonus:** Try arguing that, even though SRP-v6a is run without using TLS encrypted channel, the adversary still cannot "easily" launch offline dictionary attacks on it. Just write a simple pdf to argue it. (Hint: Using specific example is better than providing abstract explanations)

  (You can ask AI, but then you should learn its answer and write a human-friendly answer by yourself, since it is not hard to detect that a solution is written from AI)

Further Reading

- RFC document of SRCAM: https://datatracker.ietf.org/doc/html/rfc5802
- Password-Based Key Derivation Function:

  https://datatracker.ietf.org/doc/html/rfc8018#page-11
- Analysis on SRP: *Provable Security Analysis of the Secure Remote Password Protocol,* https://eprint.iacr.org/2023/1457
- Matthew Green's blog: *Should you use SRP?* *https://blog.cryptographyengineering.com/should-you-use-srp/*