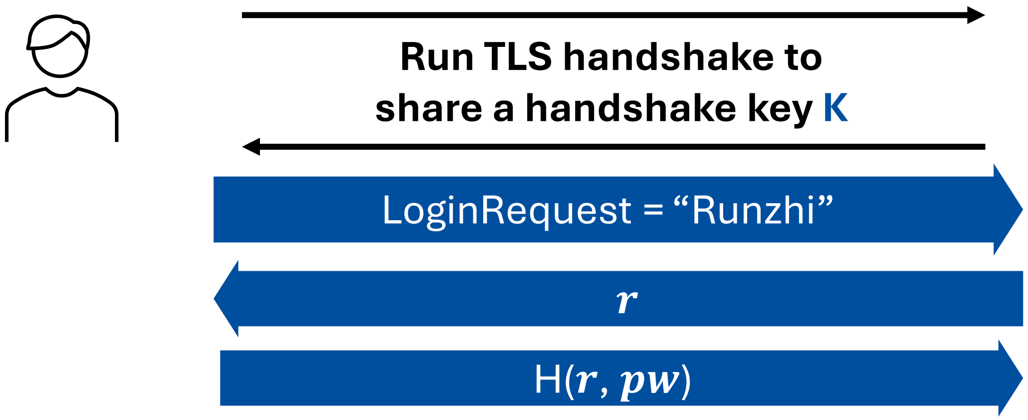
Cryptography Engineering

- 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 = pw
where pw is some string

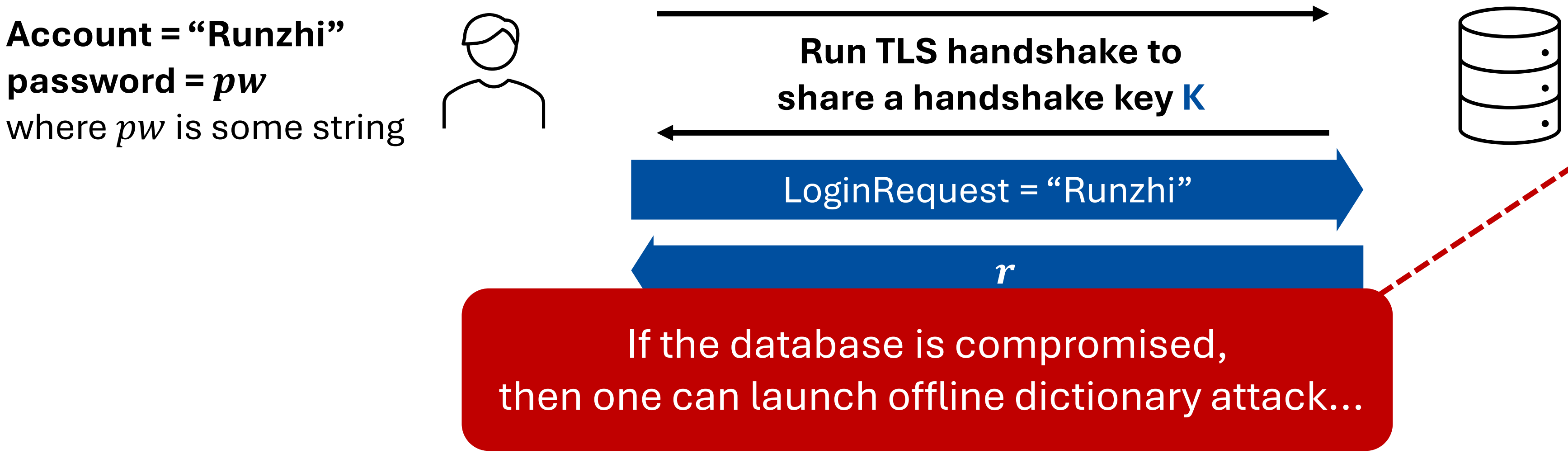


(Encrypted by the TLS handshake key K)

User	password_file
Runzhi	$r, H(r, pw)$
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

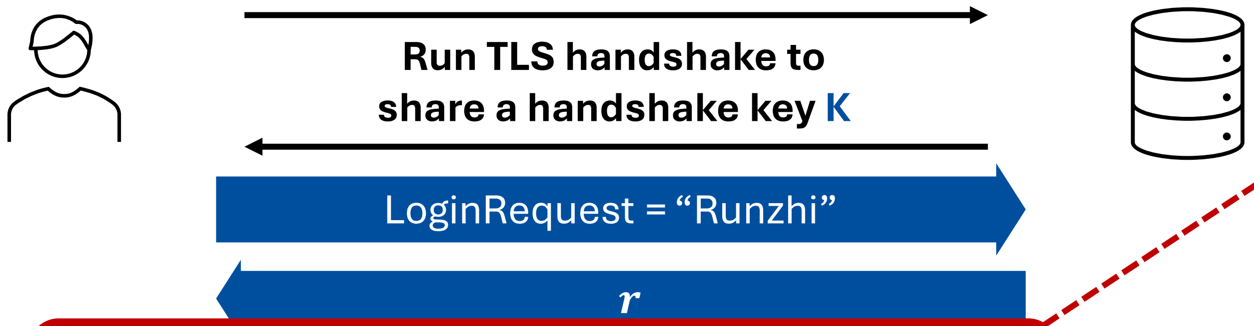


User	password_file
Runzhi	$\boldsymbol{r, H(r, pw)}$
Tom	$r_2, H(r_2, pw_2)$
…	…

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



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

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

Is it possible to increase the difficulty of offline attacks?

The SCRAM protocol

- Salted Challenge Response Authentication Mechanism

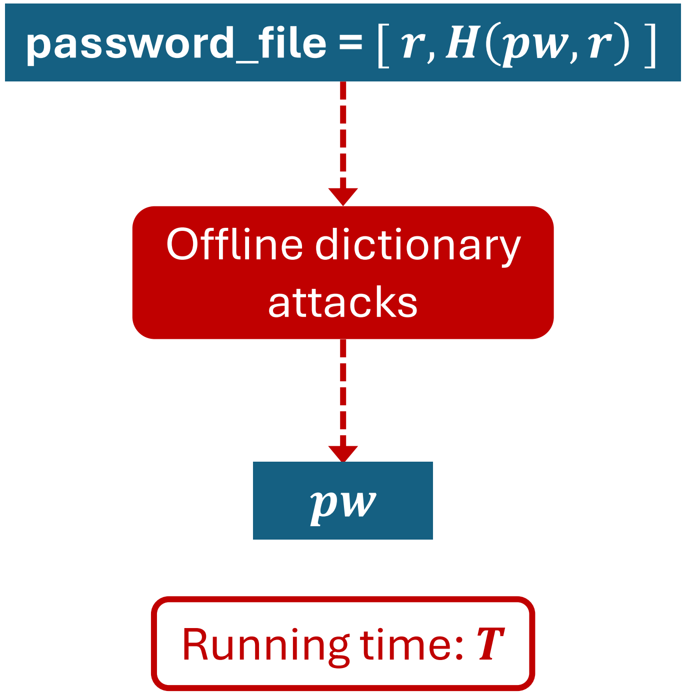
- Main idea:
  1. Add iteration in computing salted & hashed password
  2. Challenge-response Mechanism
  3. Run over TLS

- 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

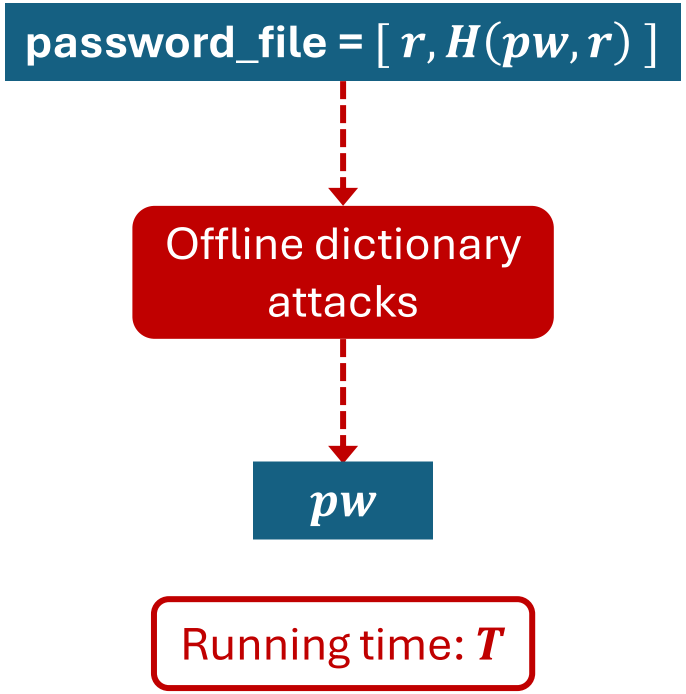
The SCRAM protocol

- Add iteration in computing salted & hashed password:



The SCRAM protocol

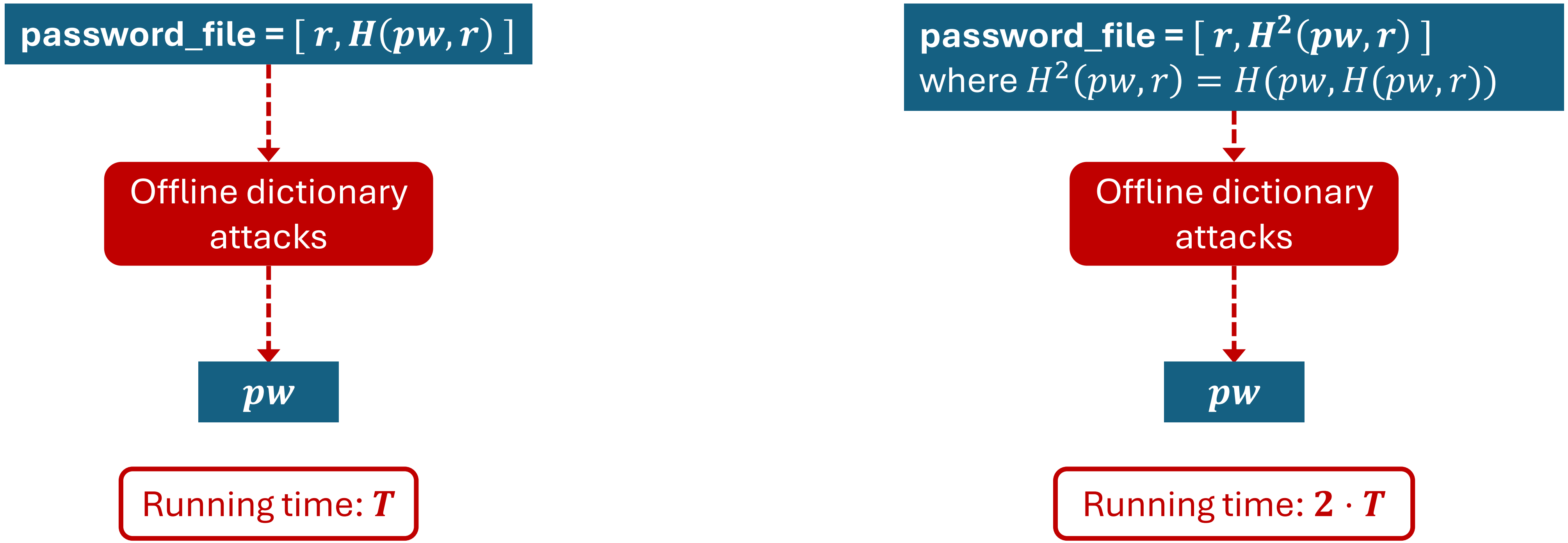
- Add iteration in computing salted & hashed password:



password_file = [$r, H^2(pw, r)$]
where $H^2(pw, r) = H(pw, H(pw, r))$

The SCRAM protocol

- Add iteration in computing salted & hashed password:



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)

$hash_1 = \text{HMAC}(pw, padded\_salt)$ // 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
$hash_i = \text{HMAC}(pw, hash_{i-1})$

Password_file = $hash_1 \oplus hash_2 \oplus \cdots \oplus hash_{\text{num\_of\_iteration}}$ // 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 $H^n(pw, r)$ = Iterate_hash_with_salt(pw, r, n)

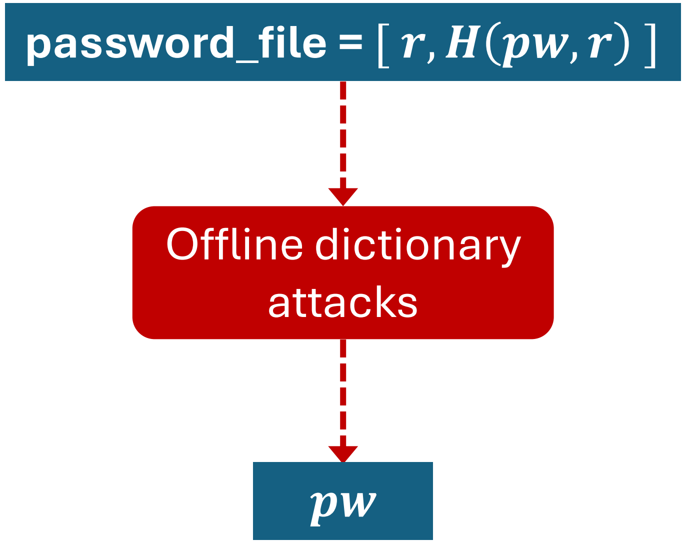
Given r, n, pw:

$$\begin{aligned} U_1 &= \mathrm{HMAC}(pw, r \,||\, \text{b'\textbackslash x00\textbackslash x00\textbackslash x00\textbackslash x01'}) \\ U_2 &= \mathrm{HMAC}(pw, U_1) \\ &\vdots \\ U_{i-1} &= \mathrm{HMAC}(pw, U_{i-2}) \\ U_i &= \mathrm{HMAC}(pw, U_{i-1}) \end{aligned}$$

We compute $H^n(pw, r) = U_1 \oplus U_2 \oplus \cdots \oplus U_{n-1} \oplus U_n$

The SCRAM protocol

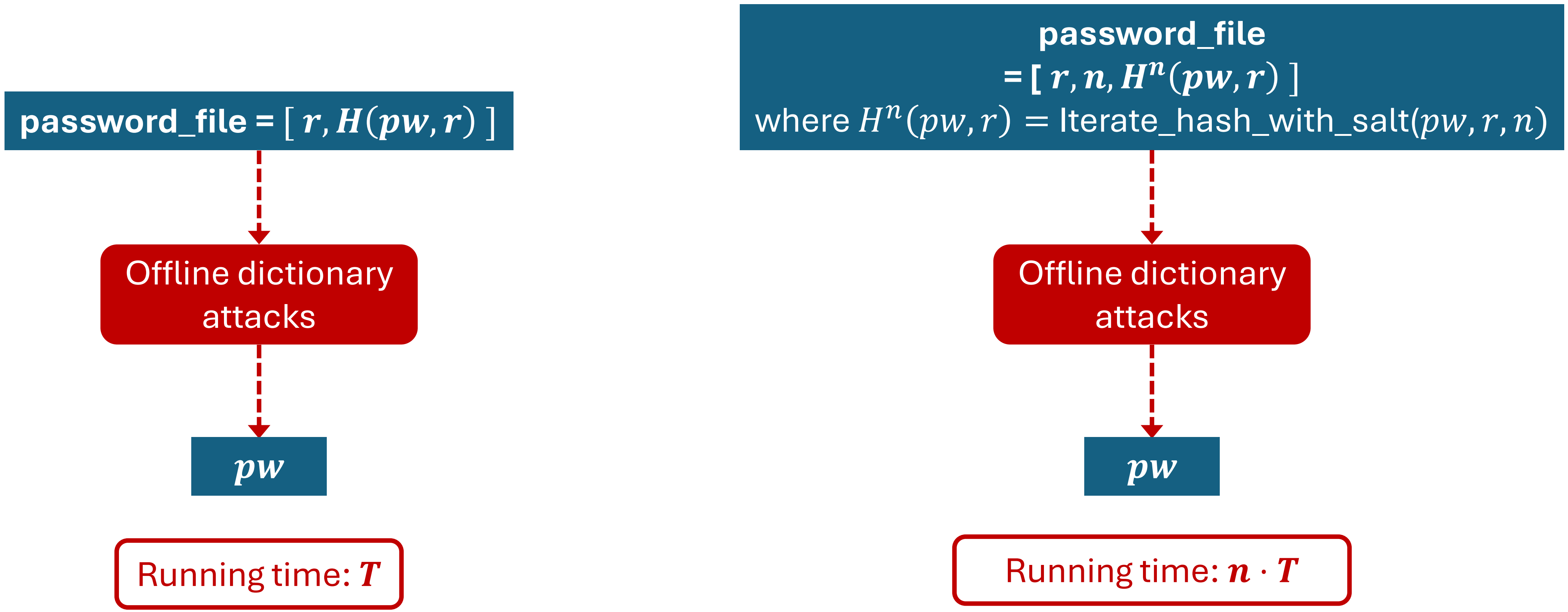
- Add iteration in computing salted & hashed password:



Running time: T

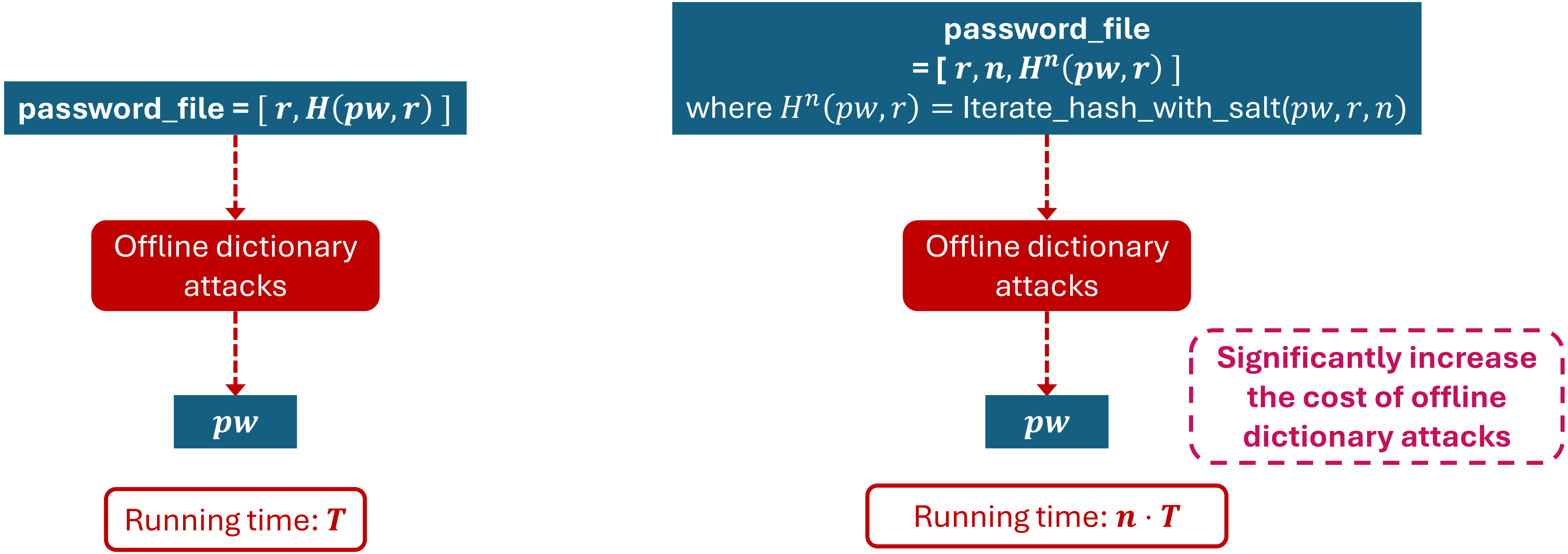
The SCRAM protocol

- Add iteration in computing salted & hashed password:



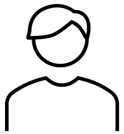
The SCRAM protocol

- Add iteration in computing salted & hashed password:

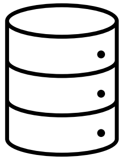


The SCRAM protocol

- Challenge-response paradigm



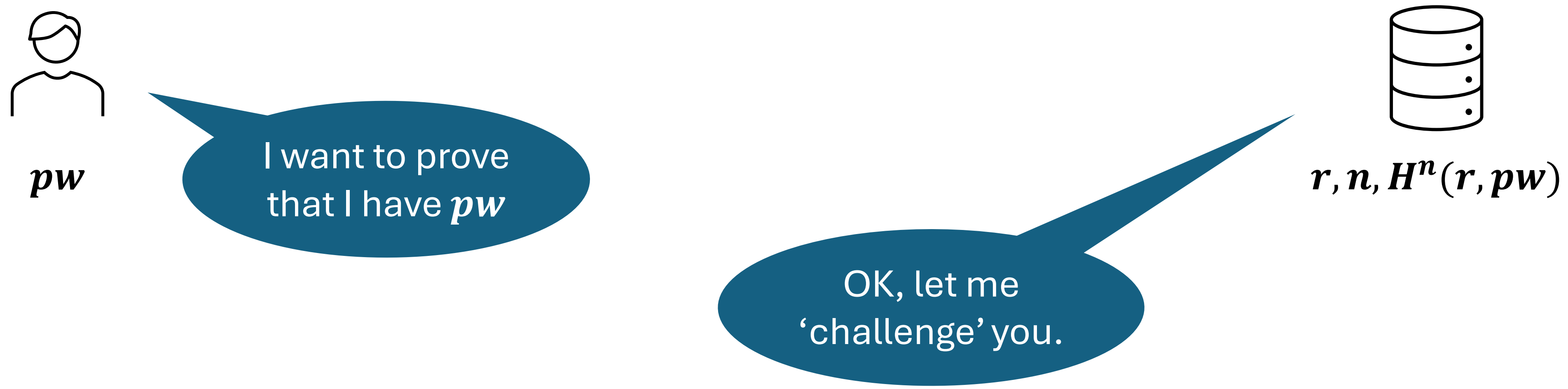
pw



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

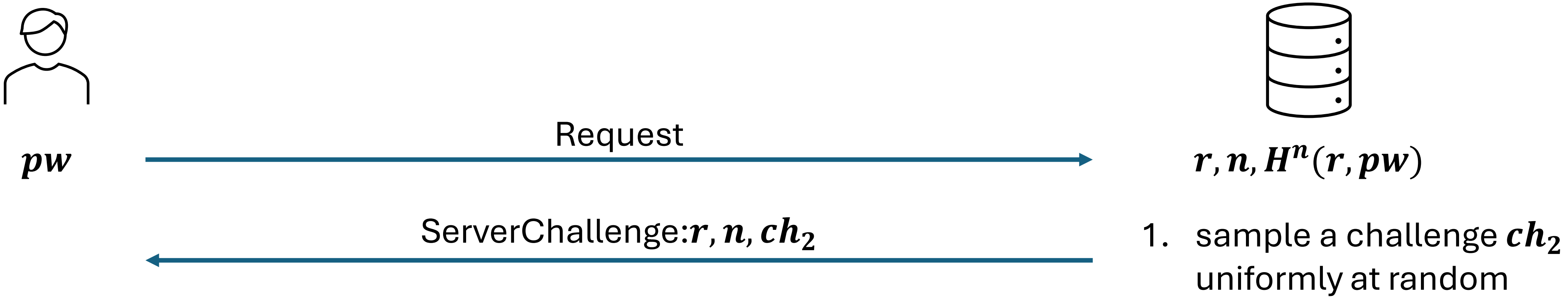
The SCRAM protocol

- Challenge-response paradigm



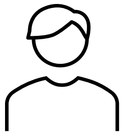
The SCRAM protocol

- Challenge-response paradigm



The SCRAM protocol

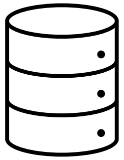
- Challenge-response paradigm



$\boldsymbol{r, n, pw}$

Request →

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



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

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*

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

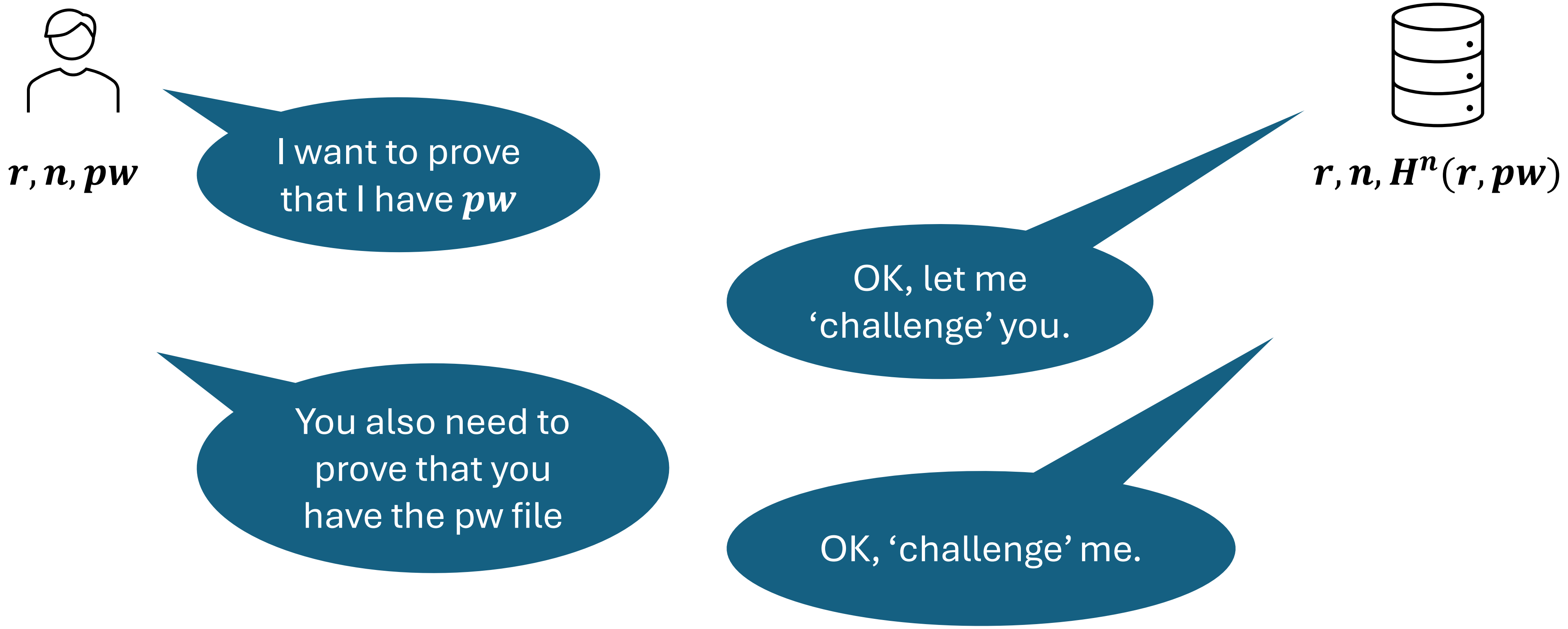
2. *Salted_pw* = $\boldsymbol{H^n(r, pw)}$
3. *Client_key* = $\mathbf{HMAC}$(*Salted_pw*, 'Client key')
4. *Auth_msg* = [Client's Name] || $\boldsymbol{r, n, ch_2}$
5. *Client_sign* = $\mathbf{HMAC}(\mathbf{H}($*Client_key*$),$ *Auth_msg*$)$ // Here $\mathbf{H}$ is the hash function used in $\mathbf{HMAC}$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

ClientProof: ***Client_proof*** ——→

6. Verify ***Client_proof***

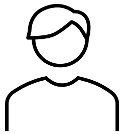
The SCRAM protocol

- Challenge-response paradigm

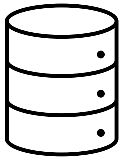


The SCRAM protocol

- Challenge-response paradigm



r, n, pw



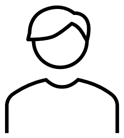
$r, n, H^n(r, pw)$

1. sample a challenge ch_1 uniformly at random

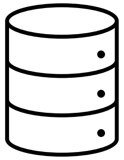
ClientChallenge: ch_1 →

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

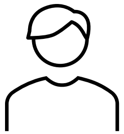
ClientChallenge: $\boldsymbol{ch_1}$ →

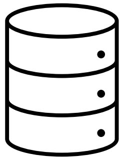
2. *Salted_pw* = $\boldsymbol{H^n(r, pw)}$
3. *Server_key* = $\mathbf{HMAC}$(*Salted_pw*, ‘Client key’)
4. *Auth_msg* = [Client’s Name] || $\boldsymbol{ch_1}$
5. ***Server_sign*** = $\mathbf{HMAC}$(*Server_key*, *Auth_msg*)

← ServerSign: ***Server_sign***

The SCRAM protocol

- Challenge-response paradigm





r, n, pw

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

1. sample a challenge ch_1 uniformly at random

ClientChallenge: ch_1 →

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

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

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

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
- Advantages: Inherent Resistance to Replay Attacks, Mutual Authentication, Channel Binding…
- Disadvantages:
  - ➢ More messages sending (i.e., higher round-trip times) , higher computation overhead (e.g., the client has to compute the iterated hash of password), …

- 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:**
  1. Higher RTTs: RTTs of TLS + RTTs of password protocol
  2. If the handshake protocol is not secure, then they can be trivially broken (leads to offline attacks immediately).

Password-based Authenticated Key Exchange

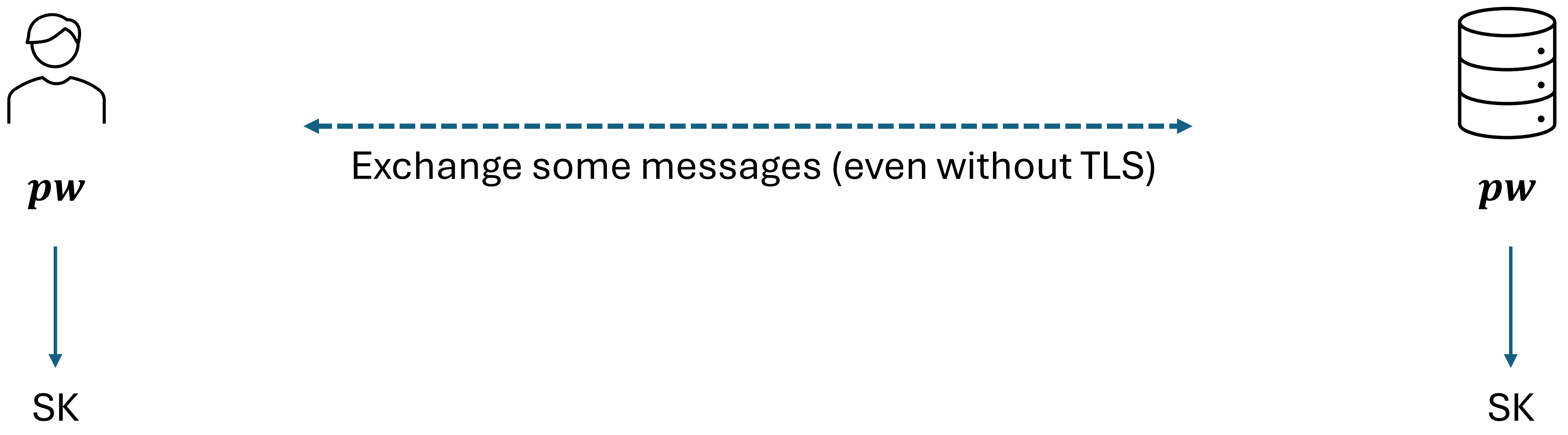
- Previous protocols: TLS + salted & hashed (& iterated) passwords
  - Advantages: Simple, rely on known constructions

- **Not entirely satisfactory:**
  1. Higher RTTs: RTTs of TLS + RTTs of password protocol
  2. 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)**

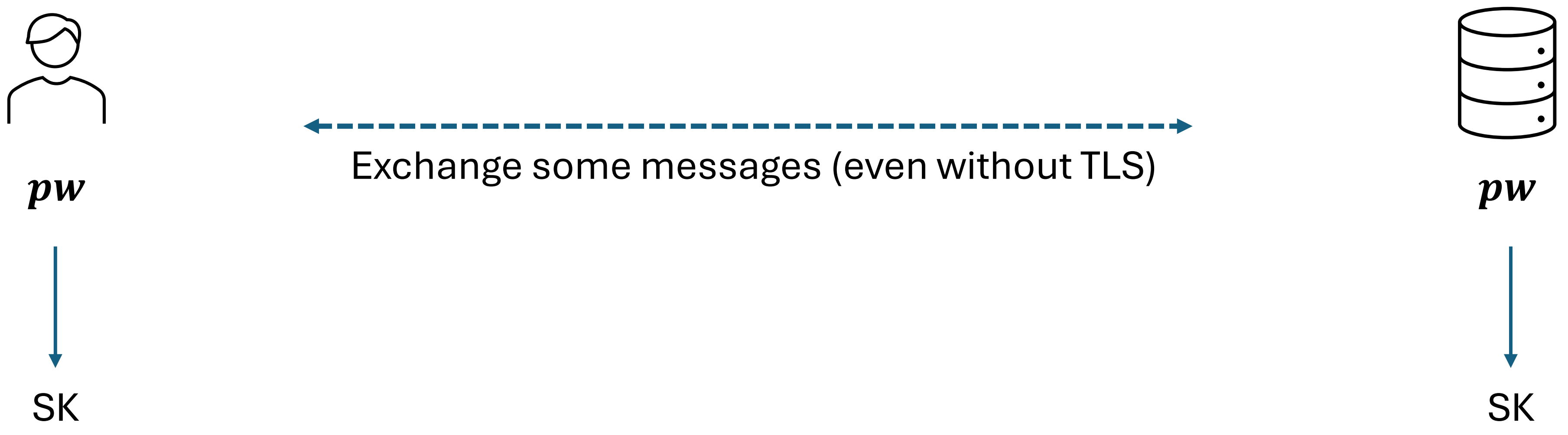
Password-based Authenticated Key Exchange

- **(Symmetric) PAKE:**



Password-based Authenticated Key Exchange

- **(Symmetric) PAKE:**

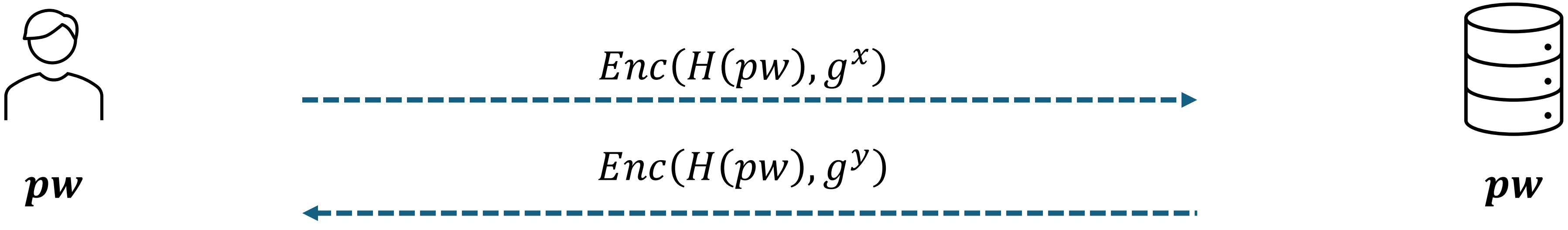


Primary Goals:

- **(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

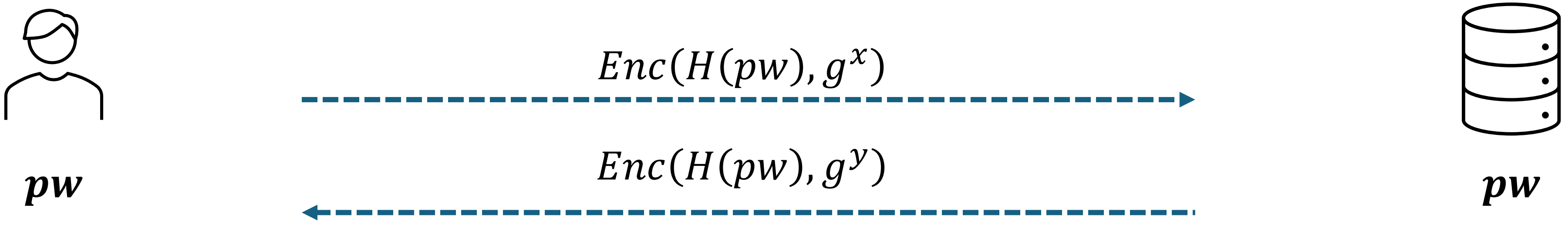


$SK = KDF(H(g^{xy}), \ldots)$ $\qquad$ $SK = KDF(H(g^{xy}), \ldots)$

- 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

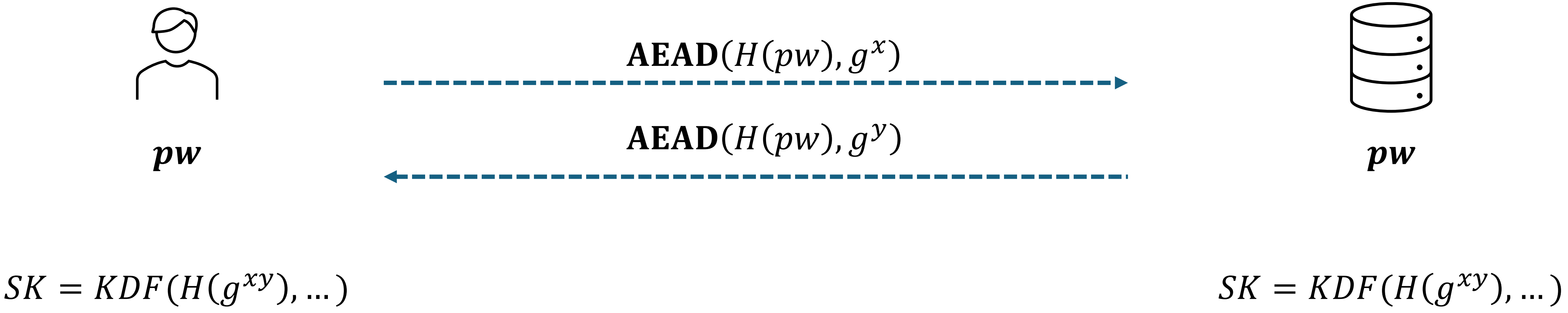


$SK = KDF(H(g^{xy}), \ldots)$ $\qquad$ $SK = KDF(H(g^{xy}), \ldots)$

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

Password-based Authenticated Key Exchange

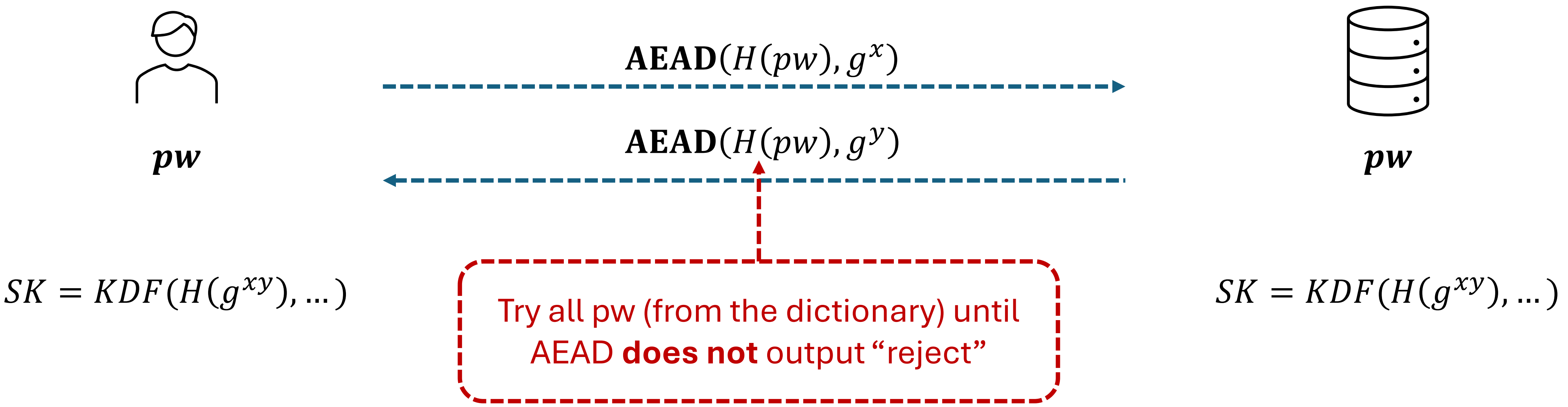
- EKE-DH protocols **based on AEAD:**



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

Password-based Authenticated Key Exchange

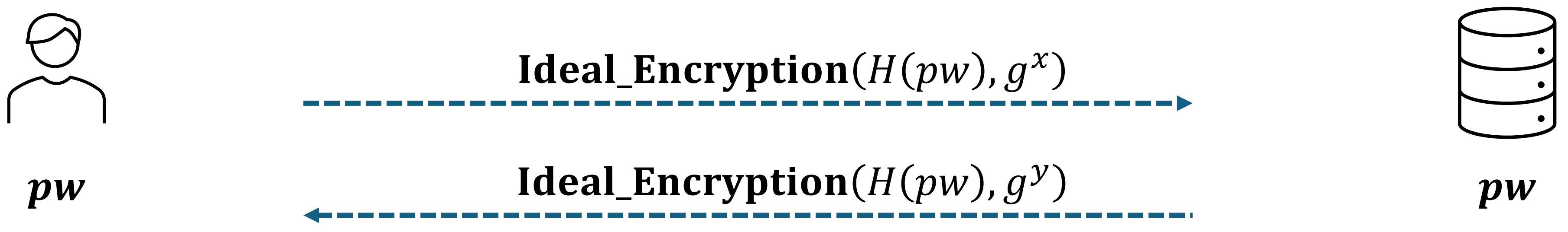
- EKE-DH protocols **based on AEAD:**



- 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:**

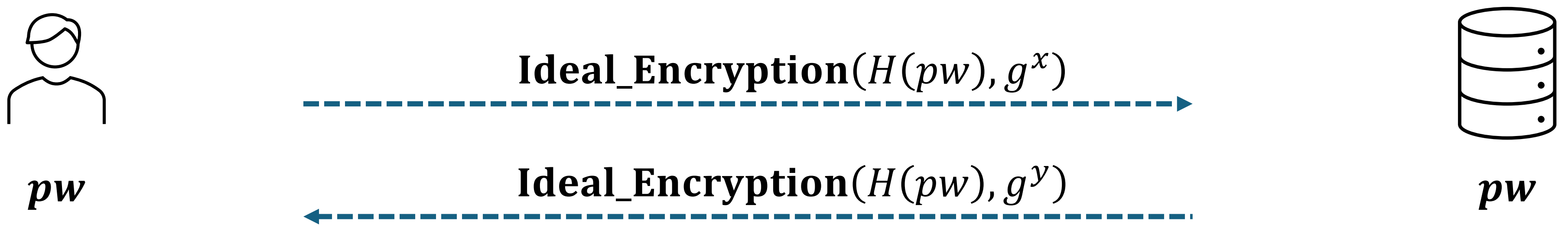


$SK = KDF(H(g^{xy}), \ldots)$ $SK = KDF(H(g^{xy}), \ldots)$

- 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:**



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

Ideal_Encryption
never (or with a negligible probability)
outputs “reject”!

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

- 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:**

pw → **Ideal_Encryption**$(H(pw), g^x)$ → pw

Ideal_Encryption$(H(pw), g^y)$ ←

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

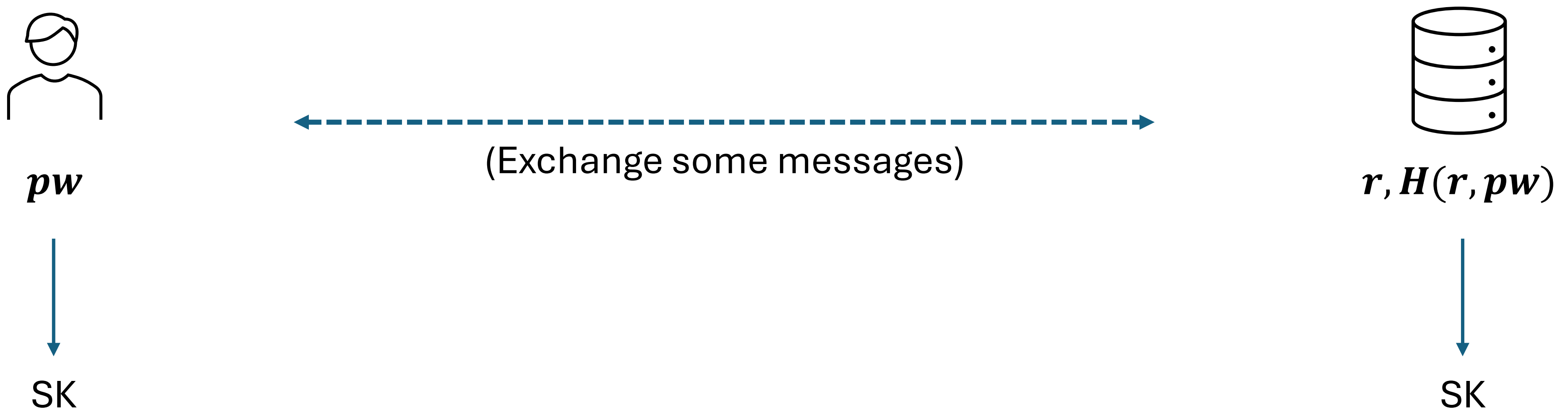
$SK = KDF(H(g^{xy}), \ldots)$ $SK = KDF(H(g^{xy}), \ldots)$

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

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

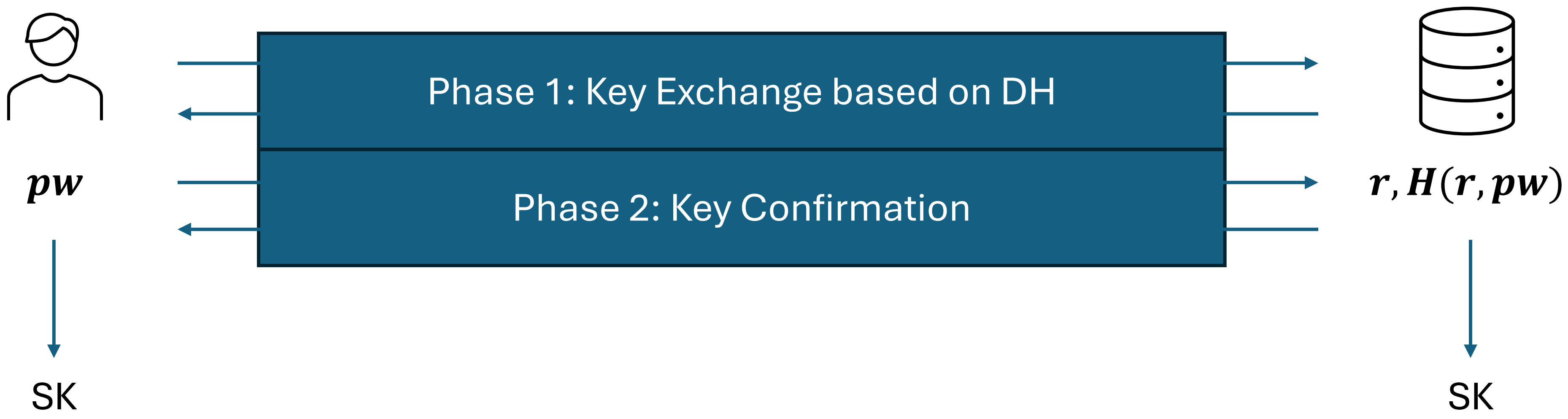
Password-based Authenticated Key Exchange

- **Asymmetric PAKE (aPAKE):**



Password-based Authenticated Key Exchange

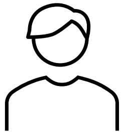
- **Secure Remote Password (SRP) Protocol**



- 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)**



$\boldsymbol{pw}$

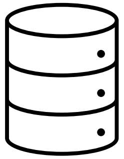
Public Parameters:

$\boldsymbol{q}$	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 $h^x \coloneqq h^x \mod N$

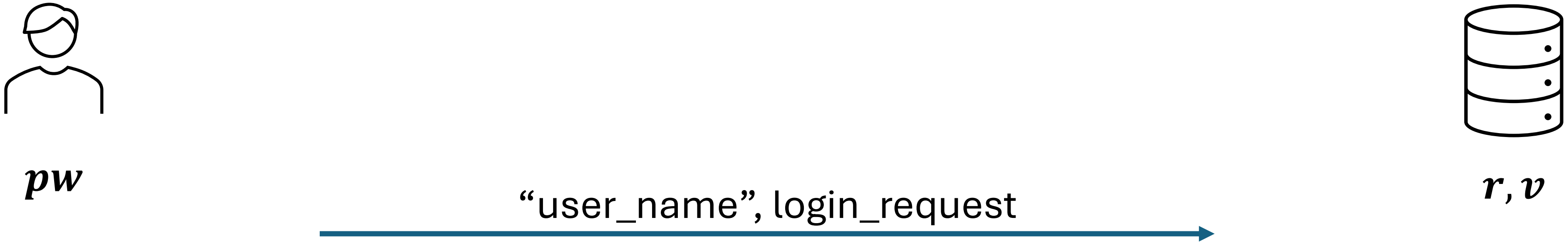


Password file:

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

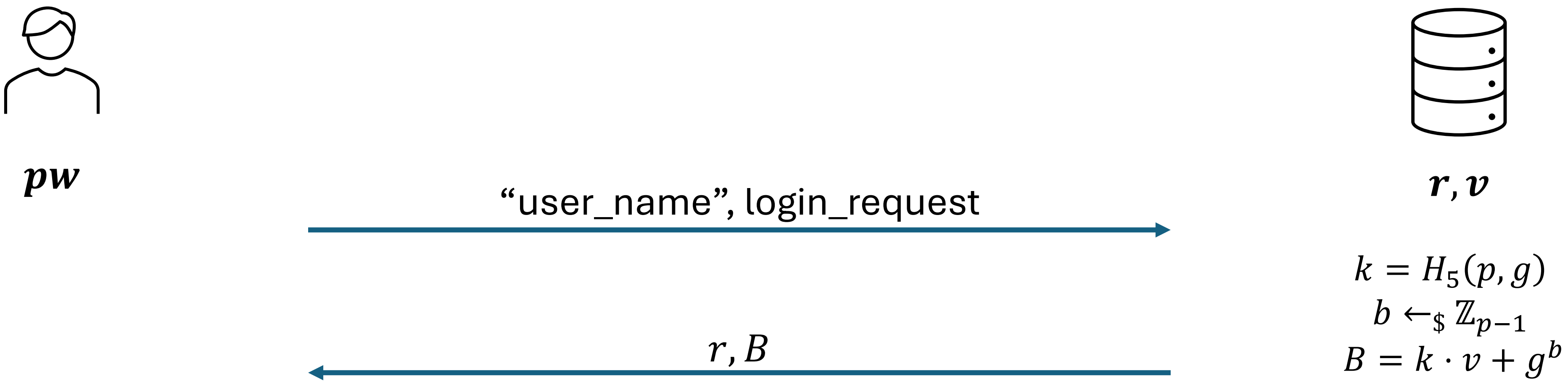
Password-based Authenticated Key Exchange

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



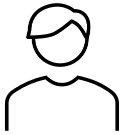
Password-based Authenticated Key Exchange

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

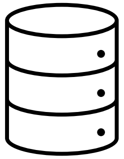


Password-based Authenticated Key Exchange

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



$\boldsymbol{pw}$



$\boldsymbol{r, v}$

"user_name", login_request →

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

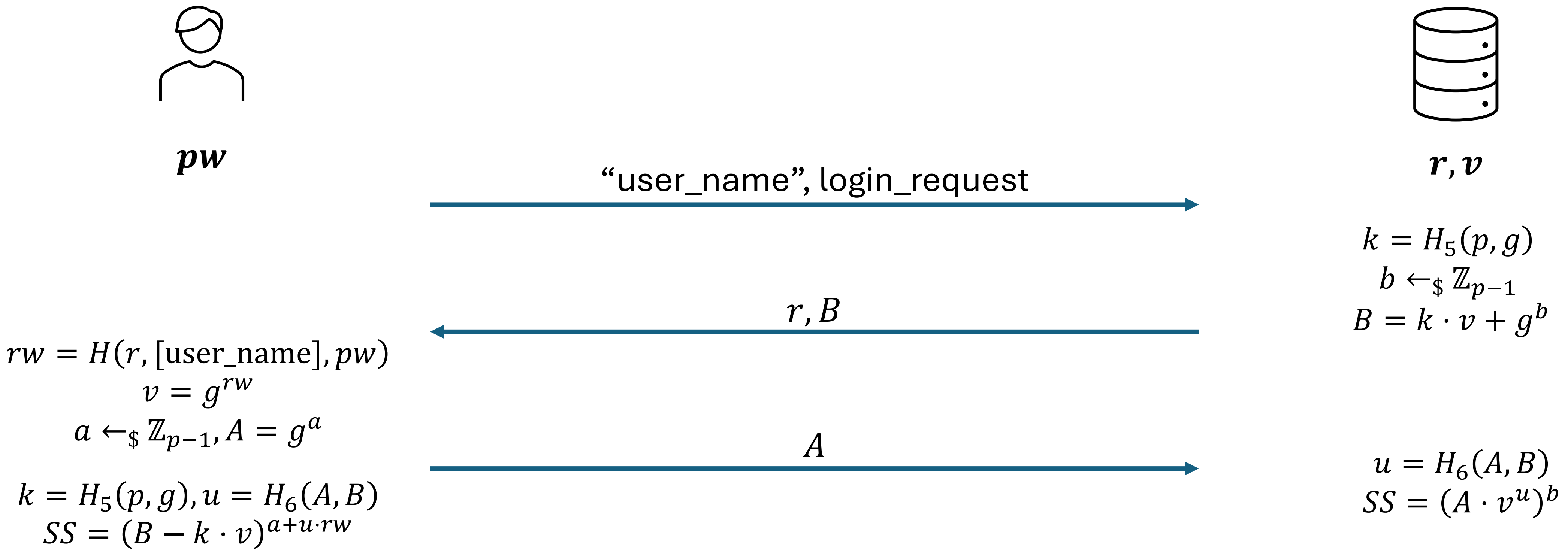
← r, B

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

A →

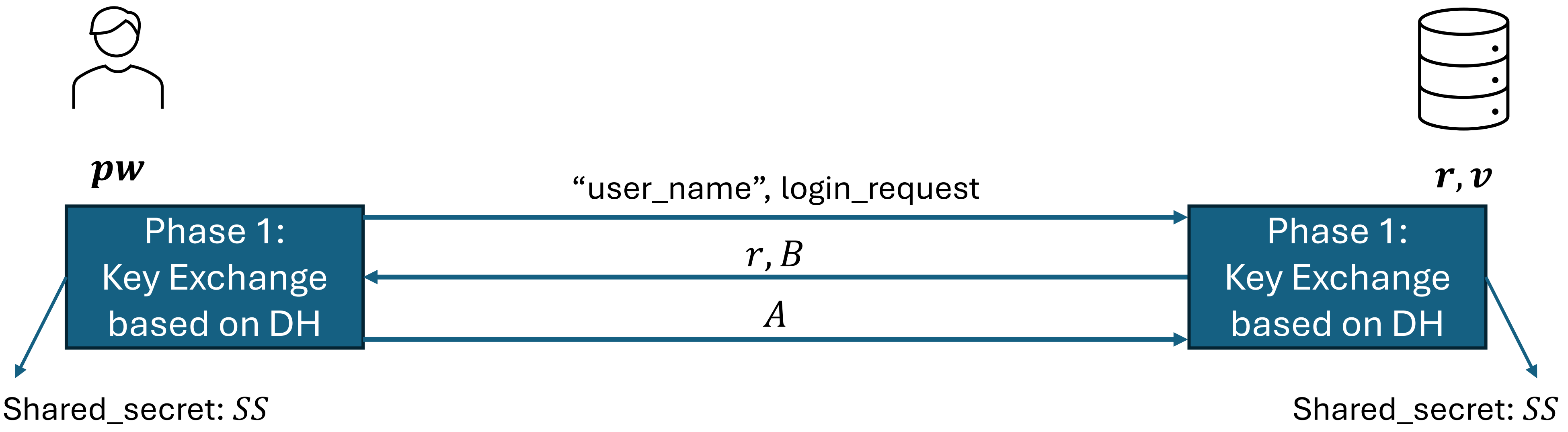
Password-based Authenticated Key Exchange

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



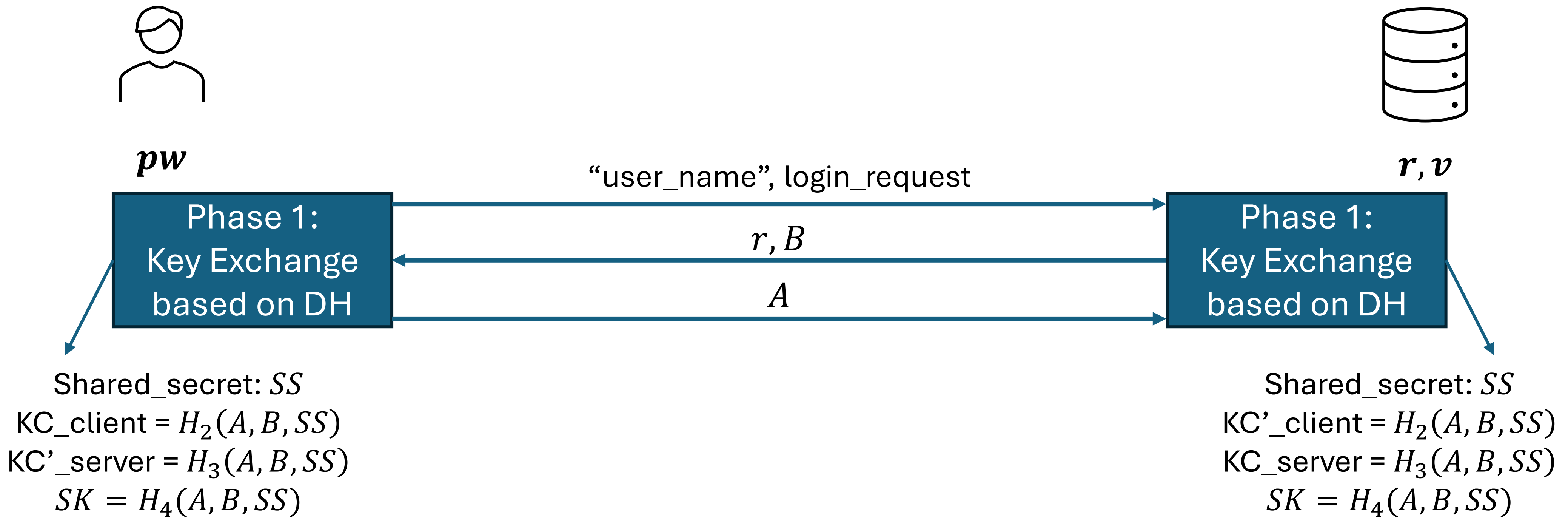
Password-based Authenticated Key Exchange

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



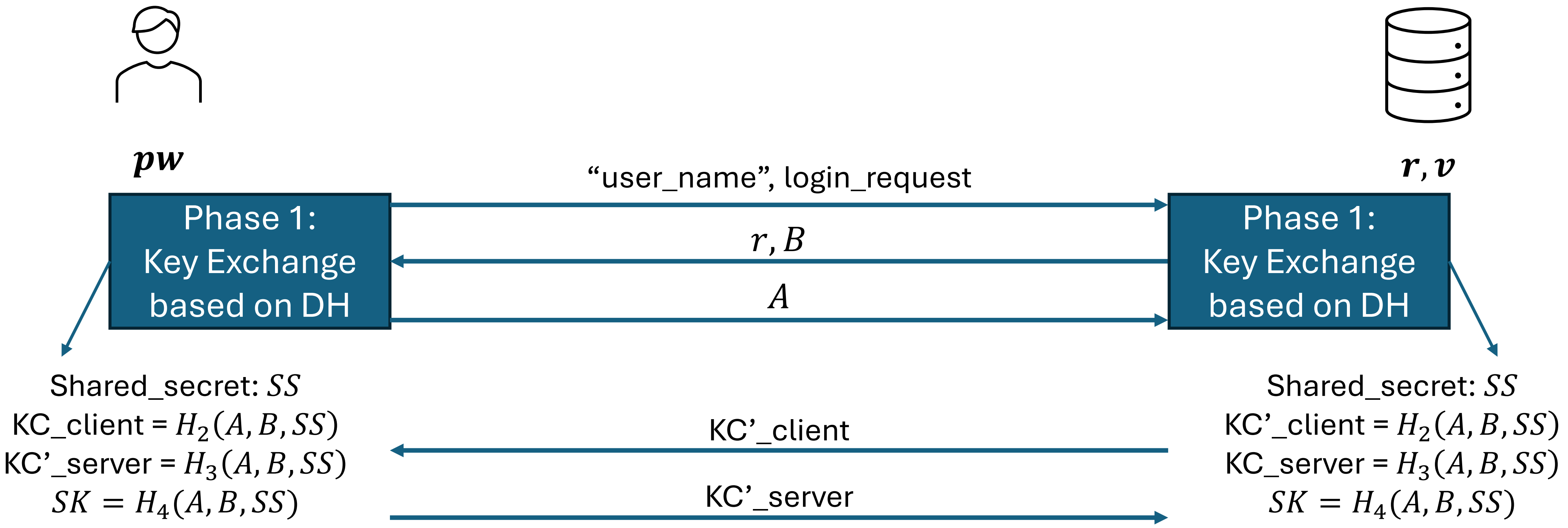
Password-based Authenticated Key Exchange

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



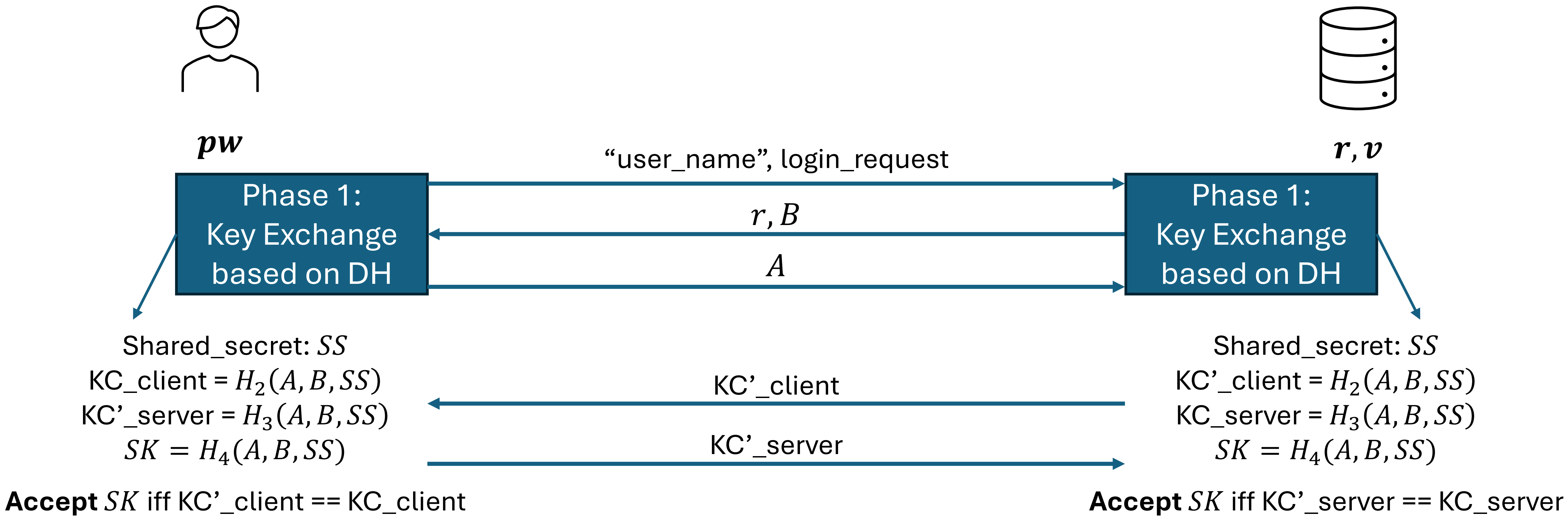
Password-based Authenticated Key Exchange

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



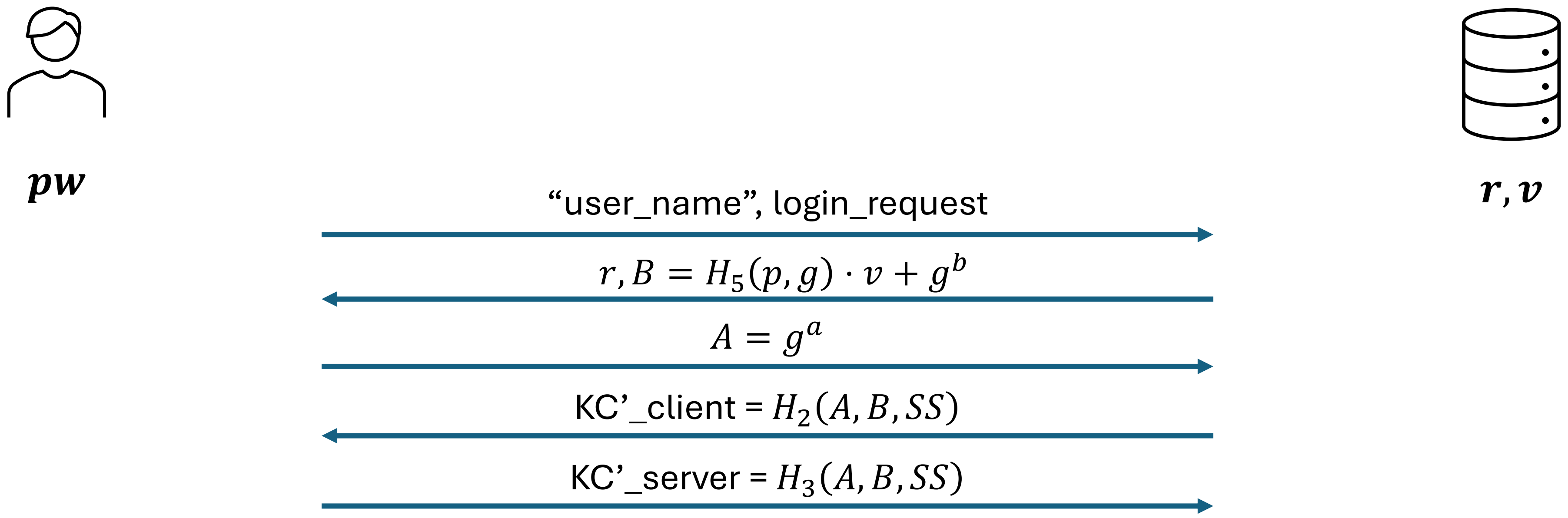
Password-based Authenticated Key Exchange

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



Password-based Authenticated Key Exchange

- **SRP-v6a:**



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)

Homework

- **Bonus (example code will be provided later):** Try implementing the following "pre-computation attacks" on SRP-v6a.
  1. Suppose that a client and a server SRP-v6a have run SRP-v6a once without using TLS (provided in the example code), and you saw the salt r and the username.
  2. Given a dictionary D (in the example code), create a new dictionary that consists of pairs

     $$(pw, v = g^{H(r, [user\_name], pw')}) \text{ for all } pw' \in D$$

  3. Suppose that now the server's pw database is compromised and you get the correct

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

  4. Using your new dictionary, recover the correct password pw^* of the client "immediately".

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/*