Basic flow of Diffie-Hellman based AKE:

Alice $x \xleftarrow{R} \mathbb{Z}_p$ $\xrightarrow{\quad g^x \quad}$ **Bank** $y \xleftarrow{R} \mathbb{Z}_p$

$k, k' \leftarrow H(g, g^x, g^y, g^{xy})$

$\sigma \leftarrow \text{Sign}(sk_{\text{Bank}}, (g, g^x, g^y, pk_{\text{Bank}}))$

$\xleftarrow{\quad g^y,\ c \leftarrow \text{Enc}_{AE}(k', (\text{cert}_{\text{Bank}}, \sigma)) \quad}$

Alice: derive $k, k' \leftarrow H(g, g^x, g^y, g^{xy})$

check σ is signature on $(g, g^x, g^y, pk_{\text{Bank}})$

under pk_{Bank} is the public key identified by $\text{cert}_{\text{Bank}}$

Bank: session key k'

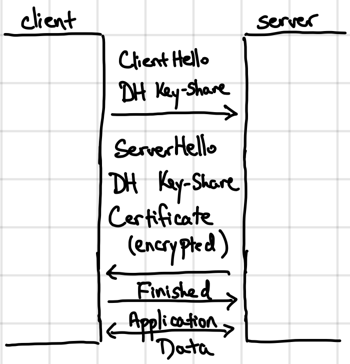
<u>intuition</u>: $\text{cert}_{\text{Bank}}$ identifies server as Bank (with pk_{Bank})

σ binds the session parameters (g, g^x, g^y) to the public key identified by $\text{cert}_{\text{Bank}}$

<u>End of protocol</u>: Alice knows she is talking to Bank (but <u>not</u> vice versa!)

"one-sided AKE" — most common mode on the web

↳ Basis of TLS 1.3 handshake ("one-sided" AKE) ALWAYS USE TLS 1.3 – Don't invent your own AKE protocol!



<u>ClientHello</u>: List of supported ciphersuites (e.g., AES-GCM-128, AES-GCM-256)

Possible TLS extensions

older systems / foreign systems may prefer different ciphers

older versions of TLS vulnerable to cipher downgrade attacks

<u>ServerHello</u>: Chosen ciphersuite

Application layer secured using unidirection keys $k_{A \to B}$ and $k_{B \to A}$

In TLS 1.3, the only long-term secret on the server is a <u>signing</u> key. This is critical for achieving forward secrecy.

<u>Forward secrecy</u>: compromise of server in the future <u>cannot</u> affect secrecy of sessions in the past

↳ In TLS, server secret is a signing key — fresh Diffie-Hellman secret used for each session is fresh ("ephemeral")

Compromising signing key allows impersonation of server, but does not break secrecy of past sessions

↳ As we will see, not all AKE protocols provide forward secrecy

Alternative (using PKE): suppose server has certificate authenticating a public key for a PKE scheme (CCA-secure):

$k \xleftarrow{R} \mathcal{K}$ **Alice** $\xleftarrow{\quad r,\ \text{cert}_{\text{Bank}} \quad}$ (nonce) **Bank** sk_{Bank}, $\text{cert}_{\text{Bank}}$

$\xrightarrow{\quad c \leftarrow \text{Enc}(pk_{\text{Bank}}, (r, k)) \quad}$

$(r', k) \leftarrow \text{Decrypt}(sk_{\text{Bank}}, c)$

check that $r' = r$

Yields statically-secure authenticated key agreement (no forward secrecy)

Compromise of sk_{Bank} compromises <u>all</u> past sessions

TLS 1.3 and authenticated key-exchange protocols on the Internet typically provide one-sided authentication (i.e., client learns id of the server, but not vice versa)

Question: how does the client authenticate to the server (without providing a certificate)
↳ e.g., how does client login to a web service?

Typical setting: client and server assumed to have some shared state [e.g., client has a password and server has an HMAC of the password]

(sk) client — (vk) server

AKE protocol $\longleftrightarrow$

client learns server's identity

identification protocol $\longleftrightarrow$

} cannot replace this with anonymous key exchange
↳ becomes vulnerable to a man-in-the-middle attack

Threat models: Adversary's goal is to authenticate to server
- Direct attack: adversary only sees vk and needs to authenticate
  (e.g., physical analogy: door lock — adversary can observe the lock, does not see the key sk)
- Eavesdropping attack: adversary gets to observe multiple interactions between honest client and the server
  (e.g., physical analogy: wireless car key — adversary observes communication between car key and car)
- Active attack: adversary can impersonate the server and interact with the honest client
  (e.g., physical analogy: fake ATM in the mall — honest clients interact directly with the adversary)

Simple (insecure) password-based protocol:

client [sk: pwd] $\xrightarrow{\text{pwd}}$ server [vk: pwd]
↓
accept if vk = pwd

NEVER STORE PASSWORDS IN THE CLEAR!

Slightly better solution: hash the passwords before storing

server maintains mappings
Alice $\mapsto H(\text{pwd}_{\text{Alice}})$
Bob $\mapsto H(\text{pwd}_{\text{Bob}})$

where H is a collision-resistant hash function

client [sk: pwd] $\xrightarrow{\text{pwd}}$ server [vk: H(pwd)]
↓
accept if vk = H(pwd)

If passwords have high entropy, then hard to recover pwd from H(pwd) [by one-wayness of H]
↳ But not true in practice...

Users often choose weak passwords (e.g., 123456, password, 123456789, ...)
↳ With a dictionary of 360 million entries, can cover about 25% of user passwords
(3% choose 123456)
(10% choose among top 25 common passwords)

} Based on password hashes that have been leaked from compromised databases

Simple hashing vulnerable to "offline dictionary attack":
adversary computes table (pwd, H(pwd)) for common passwords — completely offline
given H(pwd), can now invert with a <u>single</u> lookup if pwd is contained in the database
for LinkedIn breach in 2012, attacker stole password file with ~6 million passwords
(all passwords hashed using single iteration of unsalted SHA-1) ⟶ 90% of passwords recovered in ~6 days!

<u>Problem</u>: One-time precomputation (computing the lookup table) can be <u>reused</u> to compromise <u>many</u> passwords
Overall cost of attack: $O(m+n)$ where m is the dictionary size and n is the number of passwords to attack

<u>Defense #1</u>: <u>Salt</u> passwords before hashing: namely when storing password pwd, sample salt $\xleftarrow{R} \{0,1\}^n$ and store (salt, H(salt ‖ pwd)) on the server

typically, $n \geq 64$

<u>Note</u>: Salt is a <u>public</u> value (needed for verification)

Offline dictionary attack no longer effective since every salt value induces different set of hash values
Overall cost of dictionary attack: $O(mn)$ — need to re-hash dictionary for every salt

<u>Defense #2</u>: Use a <u>slow</u> hash function [SHA-1 is very fast — enables fast brute-force search]
- PBKDF2 (password-based key-derivation function): iterate a cryptographic hash function many times:
(or bcrypt)

$$\text{PBKDF2}(\text{pwd}, \text{salt}): \underbrace{H(H(\cdots H(\text{salt} \,\|\, \text{pwd}) \cdots))}_{\text{can use 100,000 or 1,000,000 iterations of SHA-256}}$$

honest user only needs to evaluate hash function once per authentication; adversary evaluates <u>many</u> times

<u>Drawback</u>: custom hardware can evaluate SHA-256 <u>very</u> fast
- scrypt (more recent: Argon2i): slow hash function that needs lots of <u>memory (space)</u> to evaluate
↳ custom hardware do not provide substantial savings (limiting factor is <u>space</u>, not compute)

Can also use a <u>keyed</u> hash function (e.g., HMAC with key stored in HSM)
↳ ensures adversary who does not know key cannot brute force <u>at all</u>!

<u>Best practice</u>: Always salt passwords
Always use a slow hash function (e.g., PBKDF2, scrypt) or keyed hash function or <u>both</u>!

```
$cur  = 'password'
$cur  = md5($cur)
$salt = randbytes(20)
$cur  = hmac_sha1($cur, $salt)
$cur  = remote_hmac_sha256($cur, $secret)
$cur  = scrypt($cur, $salt)
$cur  = hmac_sha256($cur, $salt)
```

raw MD5 hash – not secure!

salted, keyed hash function (key on <u>remote</u> service)

slow hash function

Facebook password onion (circa 2014)

layers gradually added over time to achieve better security (and probably to <u>avoid</u> password rehashing)