ISE Cryptography — Lecture 08

The Signal Protocol: X3DH and the Double Ratchet

What Are We Protecting?

Four ways to lose a conversation.

The Scoreboard

Attacker compromises…	Past messages	Current message	Future messages
Long-term identity key	?	?	?
Signed prekey	?	?	?
One message key	?	?	?
Full ratchet state	?	?	?

- An attacker compromises a key. ***Which messages can they read?***
- Let's build Signal one mechanism at a time.
- We'll need three mechanisms:
  - ***X3DH***, the ***symmetric-key ratchet***, and the ***DH ratchet***.
- We'll come back to this table after every section!

A Note on Names

Name	Meaning
Identity key IK	Per-user, never rotates
Signed prekey SPK	Rotated ~weekly, signed by IK
One-time prekey OPK	Used once, then destroyed
Ephemeral key EK	Fresh per session / per message
Root key RK	Seeds the next chain
Chain key CK	Advances per message
Message key MK	Encrypts exactly one message
KDF	HKDF, used everywhere below
AEAD	Encrypts with the header as associated data

- Signal's specs and Boneh & Shoup name the same objects differently.
- We use B&S terminology and gloss the Signal name on first use.
- This is the only slide with the full mapping. Worth keeping open in a new tab!

Going Asynchronous

Bob is asleep. Alice wants to text him anyway.

3DH Needs Both Parties Online

- Lecture 05 built ***3DH***: mutual authentication plus forward secrecy.
  - It needs Alice and Bob ***online at the same time*** to swap ephemerals.
- ***What breaks?*** The online assumption!
  - Bob is asleep, on a plane, out of battery.
  - Alice still wants to send ***now***.
- We can't make Alice wait for Bob to wake up.
  - We *could*, but that's terrible UX.
  - 3DH, as stated, is unusable here.

X3DH

E pluribus unum.

X3DH: Prepublish the Handshake

- Signal's ***X3DH*** (Extended Triple DH) has Bob ***prepublish*** key material to a server.
- Bob's published bundle:
  - ***Identity key*** IK_B (long-term sk): never rotates.
    - This key authenticates Bob.
  - ***Signed prekey*** SPK_B: semi-static, rotated weekly, signed by IK_B.
    - Why sign it? So Alice can verify it came from Bob, not an attacker.
    - If you can trust IK_B, then you can trust SPK_B.
  - ***One-time prekeys*** $OPK_B^{(1)}, OPK_B^{(2)}, \ldots$: a stockpile, each used once.
    - One per bundle. These can run out!
- Alice fetches a bundle, picks a fresh ephemeral EK_A, and runs ***four*** DH operations.

X3DH: The Four DH Operations

$$
\begin{aligned}
DH_1 &= \mathrm{DH}(IK_A,\ SPK_B) && \text{identity meets signed prekey} \\
DH_2 &= \mathrm{DH}(EK_A,\ IK_B) && \text{ephemeral meets identity} \\
DH_3 &= \mathrm{DH}(EK_A,\ SPK_B) && \text{base forward secrecy} \\
DH_4 &= \mathrm{DH}(EK_A,\ OPK_B) && \text{FS vs } SPK_B \text{ leak}
\end{aligned}
$$

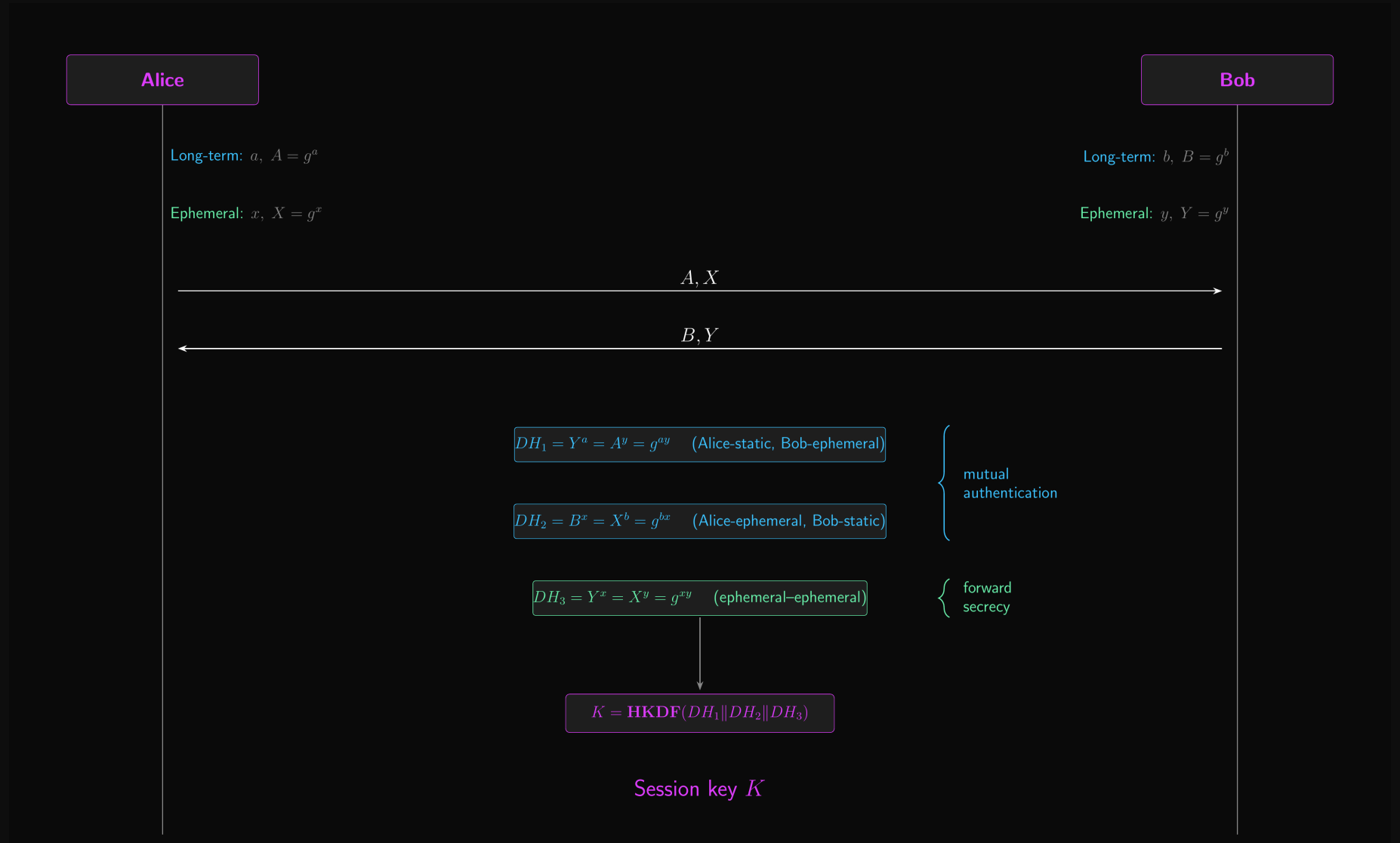
$$
SK \leftarrow \mathrm{KDF}(DH_1 \,\|\, DH_2 \,\|\, DH_3 \,\|\, DH_4)
$$

- DH_1, DH_2 provide authentication.
- DH_3, DH_4 provide forward secrecy.

Why DH_4 Exists

- SPK_B is semi-static.
  - Stealing it from Bob's device midweek gives an attacker a decent window for attacks!
- OPK_B is destroyed after a ***single*** use, so DH_4 survives an SPK_B leak.
- If the one-time prekey pool runs dry, X3DH falls back to three DHs with weaker FS.
  - Production deployments stockpile a few hundred OPKs and refill them in the background.
  - If you *do* run out, it's a good idea to warn the user that their security is degraded.
  - Lying to users about security is a bad idea.
- Alice ships EK_A and IK_A in the first message header so Bob can replay the DHs offline.

Triple Diffie-Hellman (3DH)



The Scoreboard So Far

Attacker compromises…	Past	Current	Future
Long-term identity key	safe	safe	safe
Signed prekey	safe	safe	safe
One message key	?	?	?
Full ratchet state	?	?	?

- X3DH gives Alice and Bob ***one shared secret*** SK for the conversation.
- Compromise the long-term IK alone? Past messages stay safe: the ephemerals are gone.
- Two rows down. But SK is ***one key for the entire conversation***.

One Key Is Not Enough

X3DH hands you one secret. Now send 5000 messages.

One Secret for the Whole Conversation

- SK encrypts message 1, message 2, ..., message 5000.
- ***What breaks?*** Forward secrecy (again)!
  - Compromise the device on message 5000...
  - Recover SK...
  - ...and every message back to message 1 decrypts!
- A subpoena, a seized phone, a backup leak: one secret unlocks the whole history.

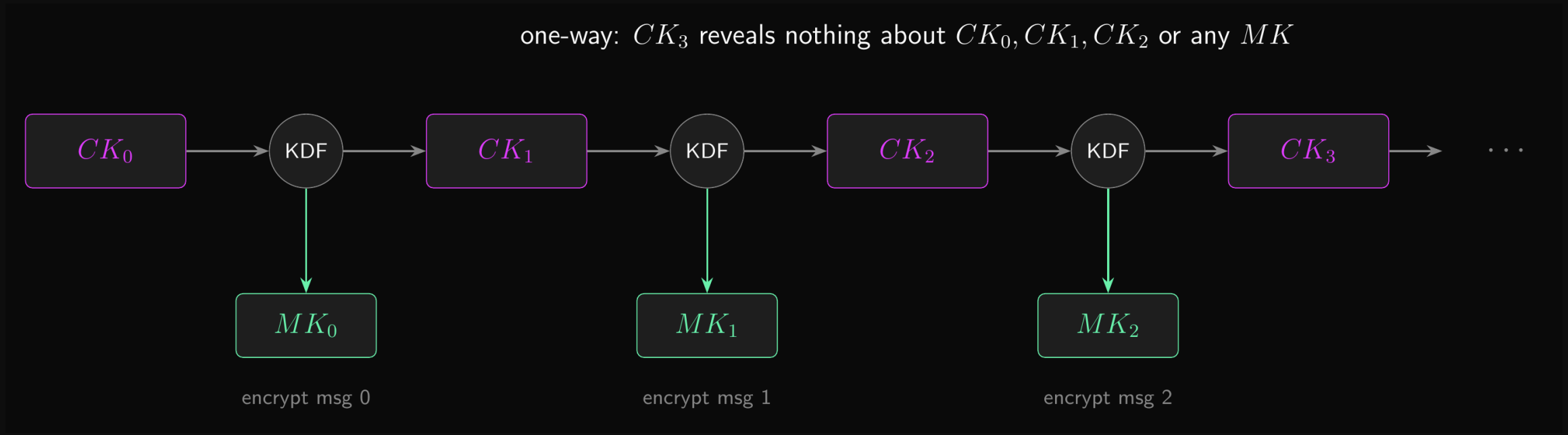
Ratchets

Ex uno plures.

The Symmetric-Key Ratchet

- Replace the single key with a ***chain***.
  - Kind of like key hierarchies (L07), but linear, not tree-structured.
- A ***chain key*** CK_i produces one ***message key*** MK_i, then advances and forgets its past
  - $(CK_{i+1},\ MK_i) \leftarrow \mathrm{KDF}(CK_i)$
  - Encrypt message i under MK_i
  - Erase MK_i and CK_i
- KDF is one-way
  - Knowing CK_{i+1} reveals nothing about CK_i or MK_i.
- What if you seize the phone at message 5000?
  - The attacker has CK_{5000}
  - But learns nothing about $CK_1, CK_2, \ldots, CK_{4999}$

The Symmetric-Key Ratchet



The Scoreboard So Far

Attacker compromises…	Past	Current	Future
Long-term identity key	safe	safe	safe
Signed prekey	safe	safe	safe
One message key	safe	that message	?
Full ratchet state	safe	exposed	?

- Compromise one message key, or the current chain key: the ***past is safe***.
- Forward secrecy: done, same as L05's ephemeral DH. Now look at the ***Future*** column.

Healing After Compromise

What if they steal the phone, then give it back?

You Can Never Break the Chain

- The symmetric ratchet is one-way. That protects the past.
  - Forward secrecy!
- ***What breaks?*** The future.
  - Steal CK_i and you can compute $CK_{i+1}, CK_{i+2}, \ldots$ forever.
  - A one-time compromise becomes permanent compromise!
- We want a new property: ***post-compromise security*** (PCS).
  - L05 forward secrecy protects the ***past***. PCS recovers the ***future***.

Ratchets 2

Electric boogaloo.

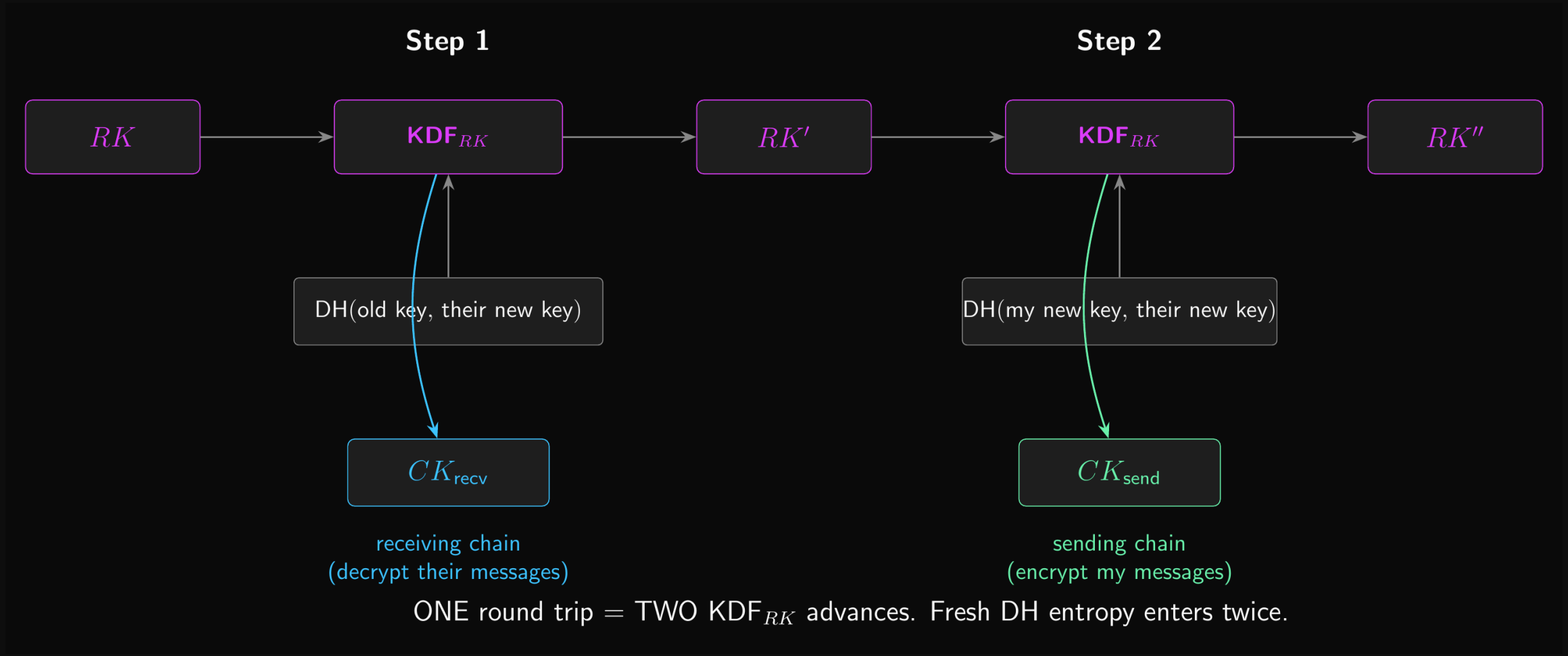
The Diffie-Hellman Ratchet

- Another ratchet?
  - This is why we call it a ***double ratchet***.
  - This one is ***asymmetric***, not symmetric.
- Mix fresh DH output into the root every time the conversation ***turns around***.
  - That is, every time Alice sends after receiving, or Bob sends after receiving.
- Each party carries a ratchet key pair. On a reply it sends a new ratchet public key.
- Receiving a new ratchet public key triggers a ***DH ratchet step***.

The DH Ratchet Step

- A single turn of the conversation runs KDF_{RK} ***twice***
- First KDF_{RK}: derive the ***receiving*** chain from their new key.
  - $(RK',\ CK_{\text{recv}}) \leftarrow \mathrm{KDF}_{RK}\big(RK,\ \mathrm{DH}(\text{my old key}, \text{their new key})\big)$
- Second KDF_{RK}: generate a fresh ratchet key, derive the ***sending*** chain.
  - $(RK'',\ CK_{\text{send}}) \leftarrow \mathrm{KDF}_{RK}\big(RK',\ \mathrm{DH}(\text{my new key}, \text{their new key})\big)$
- Two root advances per round trip.
  - Both parties contribute fresh DH entropy, so both parties heal after compromise.
- Compromise the state, steal the phone, then give it back:
  - After one round trip, the root is re-keyed with fresh DH entropy
  - The attacker has no idea what that entropy is, so they can't predict the new root, or the new chains it seeds.
  - Post-compromise security achieved!

The DH Ratchet Step: Two Root Advances



ONE round trip = TWO KDF$_{RK}$ advances. Fresh DH entropy enters twice.

The Scoreboard So Far

Attacker compromises…	Past	Current	Future
Long-term identity key	safe	safe	safe
Signed prekey	safe	safe	safe
One message key	safe	that message	safe
Full ratchet state	safe	exposed	recovers after 1 round trip

- Steal the full state now. After ***one*** undisturbed round trip, fresh DH entropy the attacker never saw re-keys the root.
- Table full! Forward secrecy ***and*** post-compromise security.
- That combination, run per message, is the ***Double Ratchet***.

Ratchet (and Clank?)

Two ratchets. Who turns when?

Two Ratchets, Different Clocks

- The symmetric ratchet advances ***per message*** (cheap, no new DH).
- The DH ratchet advances ***per round trip*** (expensive, injects entropy, gives PCS).
- ***What breaks?*** Either extreme. Symmetric-only: no healing. DH every message: needless cost, and you can't anyway, because new DH material only arrives when the other side replies.

The Double Ratchet

- A ***root chain*** sits on top. Its output seeds two symmetric chains:
  - A ***sending chain*** (CK_{send}): encrypts your outgoing messages.
  - A ***receiving chain*** (CK_{recv}): decrypts their incoming messages.
- Per message: advance the relevant symmetric chain. Per round trip: a DH ratchet step re-seeds both chains from the root.
- The trigger is purely local: ***a header carrying a new ratchet public key***.

State of One Party

- This is the basic Double Ratchet (cleartext headers). Header encryption is a named variant we do not cover.
- Each party holds: RK, CK_{send}, CK_{recv}, its own ratchet key pair, the peer's ratchet public key, and counters N_s, N_r, PN.
- Every message ships a header: $(\text{ratchet public key},\ PN,\ N)$.
  - PN = length of the previous sending chain
  - N = index in the current chain.
- The header is bound as ***AEAD associated data*** (L06): tampering is detected on decrypt.

Associated Data

$$AD = \underbrace{IK_A \,\|\, IK_B}_{\text{from X3DH}} \,\|\, \underbrace{\text{ratchet pub} \,\|\, PN \,\|\, N}_{\text{the message header}}$$

- The AEAD call on each message is $\text{AEAD}(MK,\ \text{plaintext},\ AD)$
- Identity keys IK_A, IK_B get carried over from X3DH and stay fixed for the session.
- Each piece does a specific job; remove one and something breaks.

What Breaks?

- $IK_A \,\|\, IK_B$ binds the ciphertext to ***who*** is talking.
  - Drop it and a network attacker can reroute a ciphertext into a different session whose chain happens to land on the same MK.
  - AEAD verifies, message slots into the wrong conversation.
  - This is the ***unknown key-share*** failure pattern.
- ***Ratchet public key*** binds the ciphertext to ***which DH epoch***.
  - Try to splice a message from one ratchet step onto the next?
  - Different DH pub in the AD, AEAD tag fails, decrypt aborts.

What Breaks? (cont.)

- PN, N bind the ciphertext to ***where in the chain*** it sits.
  - Reorder maliciously, replay, or rewrite a counter?
  - AD changes, tag fails.
- Net effect: AEAD's integrity guarantee extends to the routing metadata for free.
  - The receiver can ***trust the header before acting on it***.
  - The header drives every state transition the receiver makes next (which key to use, whether to start a new DH ratchet step).
    - If those bytes were unauthenticated, an attacker could steer those decisions.

When the Network Misbehaves

Message 5 before message 3. Now what?

Chains Assume In-Order Delivery

- To decrypt message 5 you need CK_5, which needs CK_3, CK_4.
- ***What breaks?*** Advancing the chain to 5 ***destroys*** MK_3, MK_4. Messages 3 and 4 can now never be read.
- Mobile networks reorder and drop constantly. A naive chain desynchronises and the conversation dies.

Skipped Message Keys

- When a message jumps ahead, derive and ***store*** the skipped message keys.
- The store is keyed by ***(ratchet public key, message number)***, not message number alone.
  - A bare N collides across a DH ratchet step; the public key disambiguates.
- On receive: try the skipped store first; else advance the chain, saving skipped keys.
- Bound the store by $\mathrm{MAX\_SKIP}$. Beyond it, ***reject***.
  - An attacker forcing unbounded skipping would exhaust memory: a denial of service. The bound is a security control, not just bookkeeping.

Worked Trace (symbolic)

$$\textbf{Alice} \to A_0 :\ RK_0 \to CK_0^s \to MK,\ \text{header}\ (P_A, 0, 0)$$
$$\textbf{Alice} \to A_1 :\ CK_0^s \to CK_1^s \to MK,\ \text{header}\ (P_A, 0, 1)$$
$$\textbf{Bob recv}\ A_0, A_1 :\ \text{DH ratchet on}\ P_A \Rightarrow RK_1,\ CK^r$$
$$\textbf{Bob} \to B_0 :\ \text{new}\ P_B \Rightarrow RK_2,\ CK^s \to MK,\ \text{header}\ (P_B, 2, 0)$$
$$\textbf{Alice recv}\ B_0 :\ \text{DH ratchet on}\ P_B \Rightarrow RK_2,\ CK^r$$

- $A_n,\ B_n$: Alice's, Bob's n-th outgoing message
- $P_A,\ P_B$: Alice's, Bob's current ratchet ***public key***
- RK_n: Root key after the n-th DH ratchet step
- $CK^s,\ CK^r$: The local party's ***sending*** and ***receiving*** chain key
- $CK_0^s, CK_1^s, \ldots$: Same chain, indexed when we want to show it advance
- MK: One-shot message key drawn from a chain
- header (P, PN, N): ratchet pub, previous-chain length, this-chain index

Deniability

Non-repudiation isn't all it's cracked up to be.

Non-Repudiation is a Footgun

- L06: a signature gives ***non-repudiation***. You ***cannot deny*** you signed it.
- ***What breaks?*** Privacy!
  - A signed transcript is courtroom evidence that you said a specific thing to a specific person.
- A whistleblower, a source, a dissident, an ISE meme account...
  - They all may want to deny that they said something.
    - Or even that they said anything at all!
  - Provable authorship is a ***threat*** in these contexts.

Deniable Authentication via DH

- X3DH authenticates with ***DH between identity keys***
  - ***Not*** signatures over the transcript.
- $\mathrm{DH}(IK_A, \cdot)$ convinces ***Bob*** that Alice is on the other end.
  - But Bob could have computed the same value himself!
  - His DH computations are indistinguishable from Alice's.
  - He can't prove to anyone else that Alice was involved!
    - He could have faked the whole transcript.
- This is ***offline deniability***: a stolen transcript proves nothing to a judge.
  - Scope it precisely. The guarantee weakens under stronger models (an online judge, key-compromise scenarios). It is deniability against an ***offline*** transcript, not a universal disappearing act.
- Same DH primitive as L05, opposite goal to L06's signatures.

Groupchats

How can we scale a double ratchet?

Pairwise Sessions

- Run a Double Ratchet between ***every pair*** in the group.
- A group of n members needs $\binom{n}{2}$ sessions.
- To send one message to $n - 1$ recipients:
  - Encrypt it $n - 1$ ***times***, once per peer session.
  - No savings whatsoever…
- ***What breaks?*** Scalability.
  - You inherit every property of the Double Ratchet, per pair.
  - But a 200-person group means 199 ciphertexts every time someone says “lol”.
- iMessage used to use this approach.

Sender Keys

- Each sender holds ***one symmetric chain*** for the whole group: a ***sender key***.
- Distribute it ***once*** to every member via the pairwise Double Ratchets.
  - The distribution is $O(n)$, but amortises over the lifetime of the key.
- To send: advance the symmetric ratchet, encrypt under MK_i, ***broadcast one ciphertext***.
  - $O(1)$ work and ciphertext per message.
- ***What breaks?*** Post-compromise security ***within the group***.
  - The sender key has no DH ratchet. Steal it, read everything that sender sends until they rotate.
  - Rotation means another $O(n)$ redistribution round.
- ***Removing a member*** is uglier still:
  - Everyone left has to generate a fresh sender key
  - Then redistribute it through their pairwise sessions
  - ...making sure the ex-member never sees it!

MLS and TreeKEM (RFC 9420)

- ***Messaging Layer Security*** (2023) does group keying properly.
- Members sit at the leaves of a binary tree. Each interior node has its own key pair.
  - A member knows the secret keys for every node on the path from their leaf to the root.
- A ***commit*** updates one member's path:
  - Regenerate node secrets from leaf to root.
  - Eencrypt each new secret to the public key of the ***sibling node*** at that level.
  - The members in that sibling subtree already share the sibling node's key. They can decrypt!
- ***Cost per update:*** $O(\log n)$ ***ciphertexts.*** A 1000-person group is ten, not nine hundred and ninety-nine.
- ***Continuous Group Key Agreement*** (CGKA). Every commit injects fresh entropy into the group secret: ***forward secrecy*** and ***post-compromise security***, at the group level, by design.
- Adds and removes are just commits that change the tree's shape.

Post-Quantum Security

Abandon hope, all ye who...

What Breaks?

- The Double Ratchet rests on symmetric KDFs and AEAD: quantum-resilient.
- ***What breaks?*** X3DH. It rests on ***Diffie-Hellman***...
  - ...and Shor's algorithm breaks DH.
- An adversary records X3DH transcripts today and decrypts them once a quantum computer exists: ***harvest now, decrypt later***.
  - The ratchet cannot save a session whose seed came from broken DH.

PQXDH

- Signal's ***PQXDH*** (2023) bolts a post-quantum KEM onto the X3DH handshake.
  - Classical DH ***and*** an ML-KEM encapsulation both feed the root.
  - An attacker must break both to recover the root, so the handshake is post-quantum secure.
- We're not covering that today, though!

Conclusion

What did we learn?

The Scoreboard, Filled In

Attacker compromises…	Past	Current	Future
Long-term identity key	safe	safe	safe
Signed prekey	safe	safe	safe
One message key	safe	that message only	safe
Full ratchet state	safe	exposed	recovers after 1 round trip

- ***X3DH*** filled the top two rows: asynchronous, authenticated, deniable session setup.
- The ***symmetric ratchet*** gave forward secrecy (the Past column).
- The ***DH ratchet*** gave post-compromise security (the Future column).
- Together: the ***Double Ratchet***.
- No new primitives! Just a clever combination of tools we already had.

A Note on Implementation

- For the sake of your EPIC grades, ***don't*** implement this yourself!
  - There's a reason we have libraries like libsignal and libsodium.
- If you're using this in the EPIC, use a battle-tested library.
  - You need to explain what properties it gives you and why.
  - And you need to use it correctly (and more importantly, be able to convince me).
  - But you don't need to implement it yourself. That's a recipe for disaster.
- ***Don't roll your own crypto***, and that includes protocols!

For Next Time

- Optional reading:
  - Signal: The X3DH Key Agreement Protocol
  - Signal: The Double Ratchet Algorithm
  - The PQXDH Key Agreement Protocol
- Next lecture: ***Post-Quantum Cryptography*** – ML-KEM, ML-DSA, and the hybrids that make PQXDH possible.

Questions?

Ask now, catch me after class, or email eoin@eoin.ai