TREE-BASED APPLICATION KEY SCHEDULE

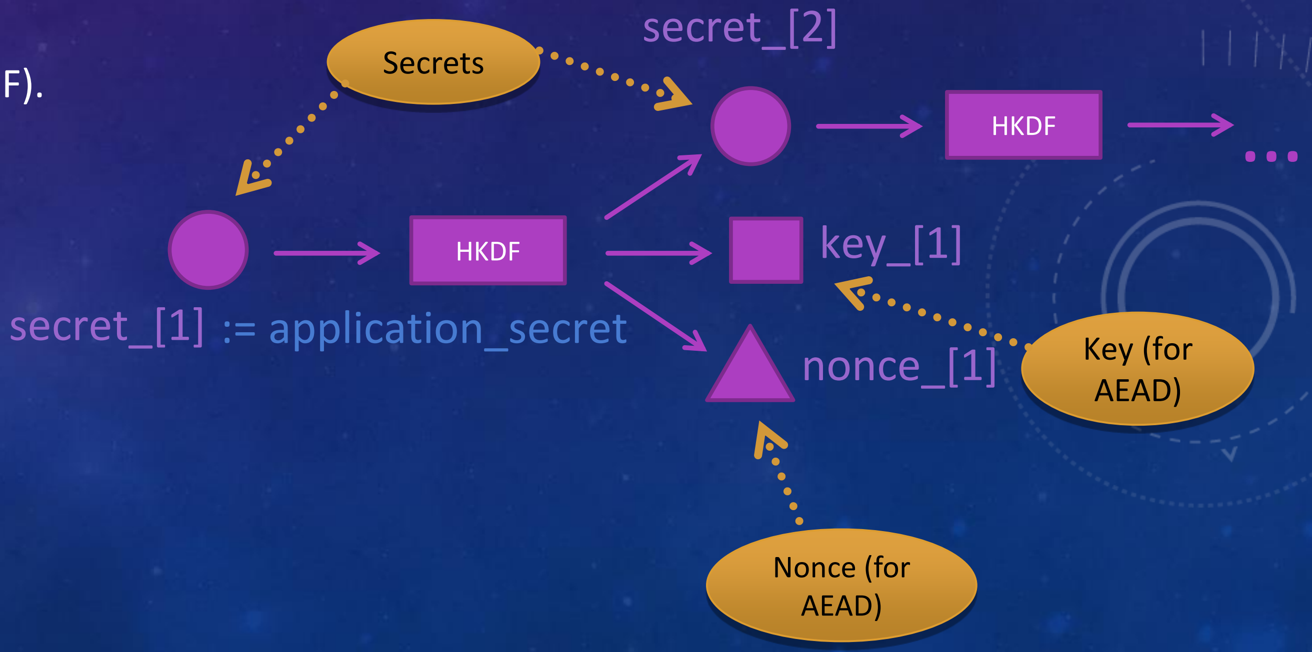
JOËL ALWEN – WICKR

OVERVIEW

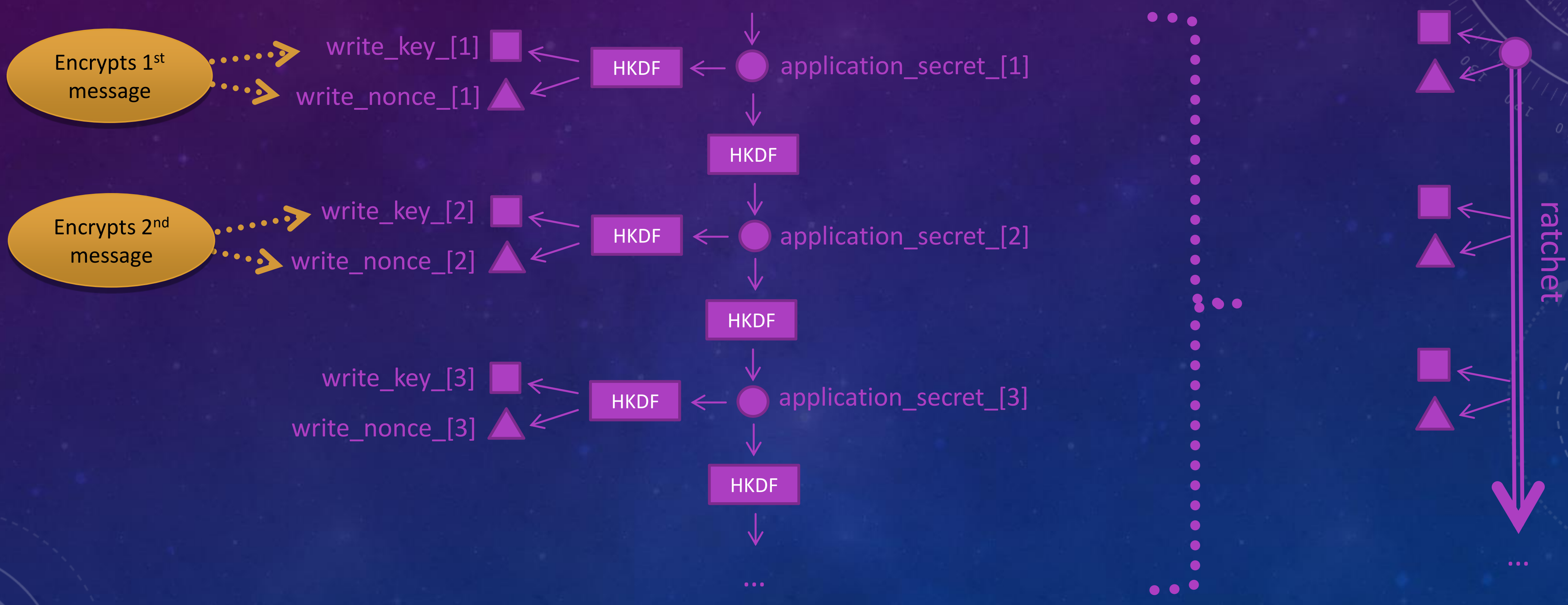
I. Key & Deletion Schedules
II. Current Application Key Schedule: Sender Ratchets
III. Proposed Application Key Schedule: Tree-Based
IV. Discussion / Open Questions

DELETION SCHEDULE: GENERAL KEY SCHEDULE

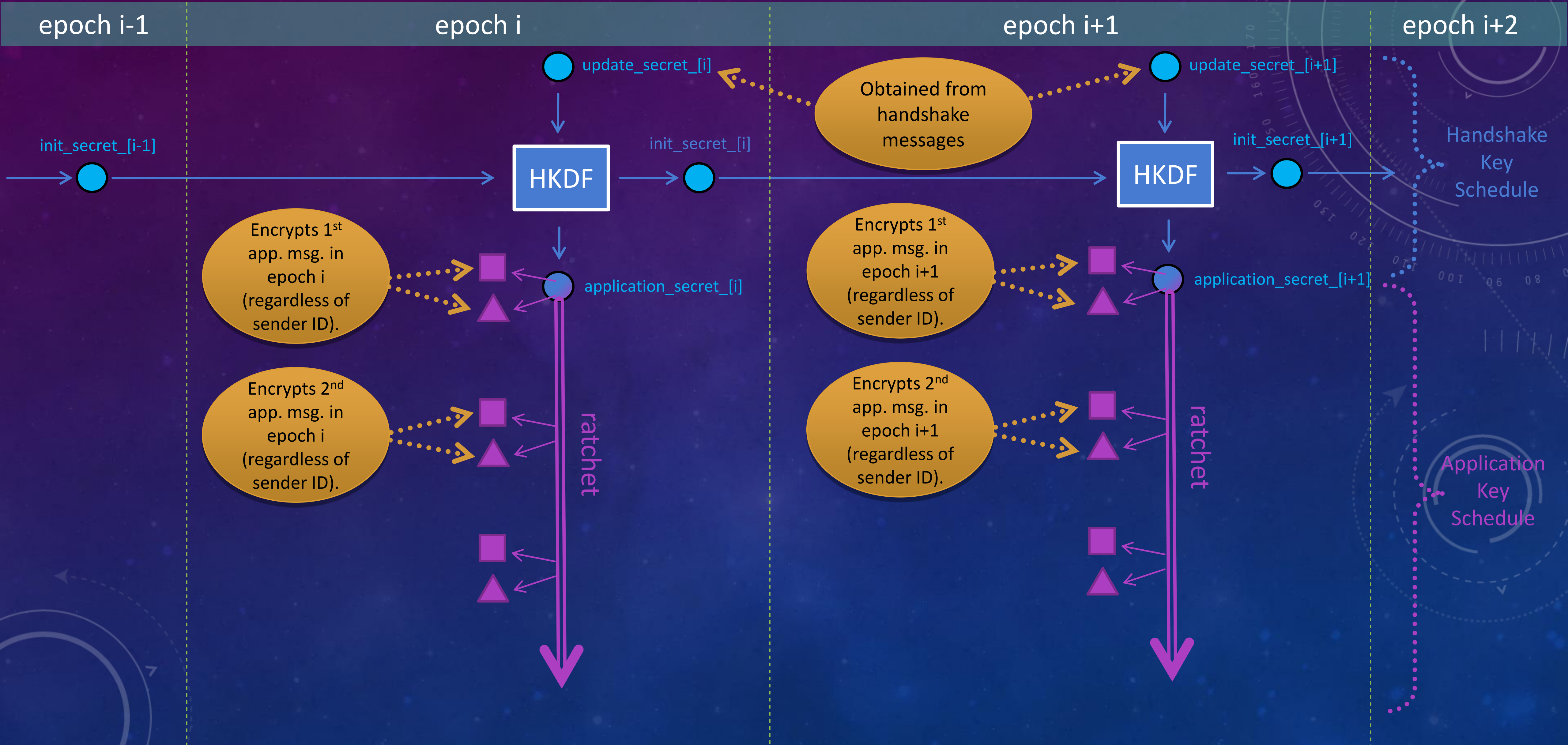
- Terminology: A “Value” is either a Secret, Key or Nonce.
- A Key Schedule is a tree of values:
  1. Each node assigned one value.
  2. Child’s value derived from parent’s value (using HKDF).
  3. Root value = application_secret.
  4. Leaf values are either Key or Nonce.
  5. Internal values are all Secrets.



SYMMETRIC RATCHET



APPLICATION KEY SCHEDULE: GROUP RATCHET



DELETION SCHEDULE: CONSUMED VALUES

- Forward Secrecy Goal: Future compromises don't harm security of today's communication.
- Methodology: Use key material 1 time only and delete as soon as used.

- Terminology: A Key or Nonce value are called "consumed" if they were used either to:
  - Encrypt a message for sending.
  - Decrypt (successfully) an incoming message.
- Terminology: A (Secret) Value is "consumed" if (at least) one of its children is consumed.
  - Why? Not enough to delete a key/nonce if we store another value that lets us re-derive that key.

- Deletion Schedule: "Delete all values the moment they are consumed."

KEY & DELETION SCHEDULE: GENERIC ALGORITHM

Nodes have 2 Boolean flags (initialized to false).

- *consumed*.
- *stored*.

Leaves have unique ID L

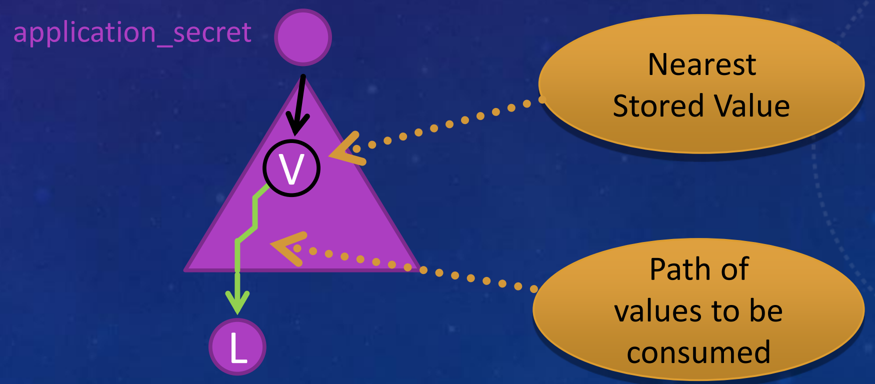
- e.g.   L = [key/nonce, sender, msg#]

Key Schedule : Init()

```
1. root.value := application_secret
2. root.stored := true
```

Key Schedule: getValue(L)

1. Find nearest *stored* ancestor V.
2. "Consume" values on path back down to L:
3. temp := L.value
4. Delete L.value
5. L.stored := false
6. L.consumed := true
7. Return temp.



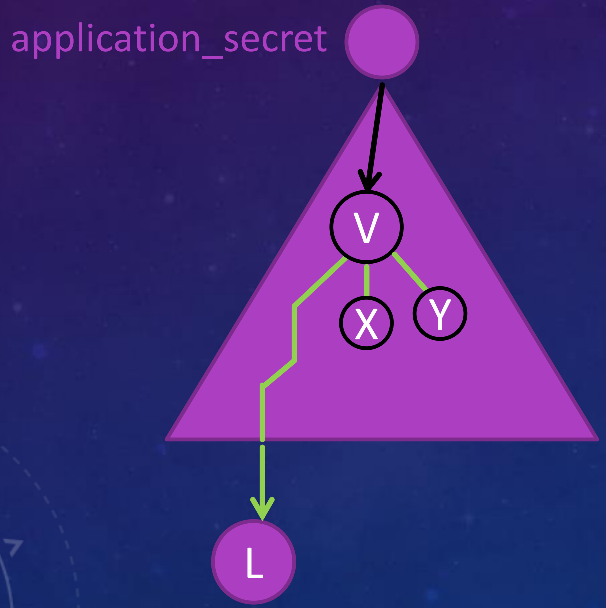
What if Decryption fails?

- Option 1: Only apply changes to key schedule state if decryption succeeds
- Option 2: Always apply changes to state except only delete & consume L.value if decryption succeeds.

KEY & DELETION SCHEDULE: GENERIC ALGORITHM

"Consume" a Value?

- For PFS → Must delete Value.
- So first derive & store all (un-consumed) children!



Key Schedule : Consume(V)

```
1. For all U = V.child do:
   a) If (U.consumed = false) && (U.stored = false) then:
      i.  U.value := HKDF(V.value,…)
      ii. U.stored := true
2. Delete V.value
3. V.stored := false
4. V.consumed := true
```

Key Schedule: getValue(L)

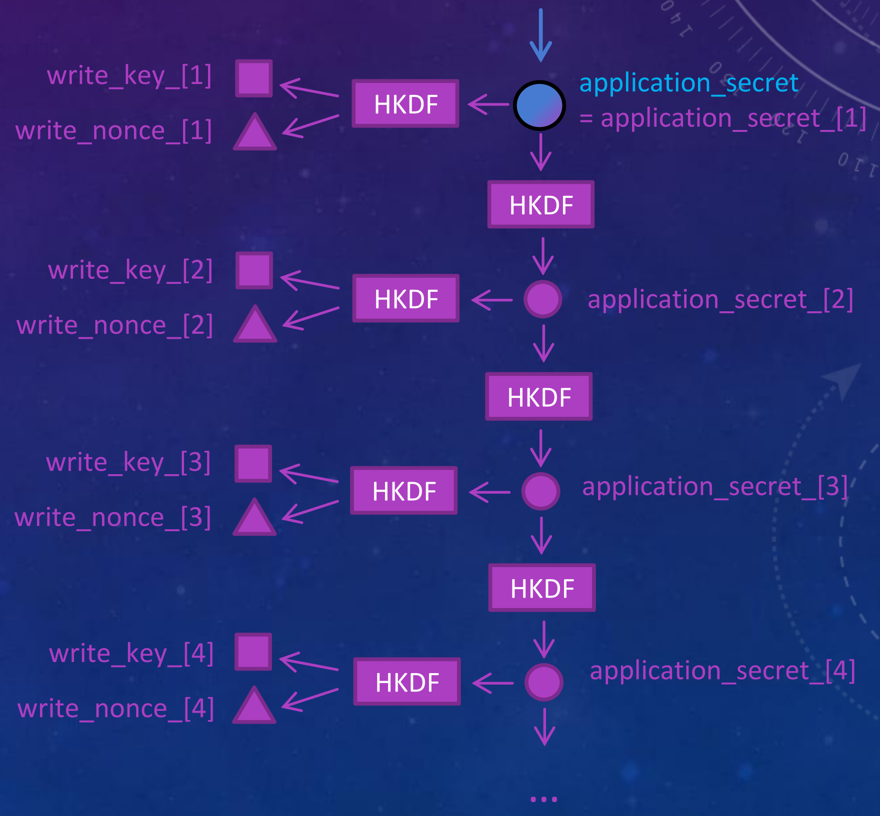
```
1. Node V := Leaf(L)
2. While V.stored = false:
   a) V := V.parent
3. For nodes U on path V → Leaf(L).parent:
   a) Consume(U)
4. return_value = L.value
5. Consume(L)
6. Return return_value
```

PRO/CON FOR GROUP RATCHET

Pro: very efficient deletion schedule.

Con: sending collisions possible!
Requires re-encrypt & re-send.

- Could lead to starvation.
- Opens up new issues: how to signal that a collision occurred? When to be sure collision did not occur? How to avoid splitting the group? Security? Etc.



APPLICATION KEY SCHEDULE: SENDER RATCHETS

Current Application Key Schedule: “Sender Ratchets”

Basic Idea:

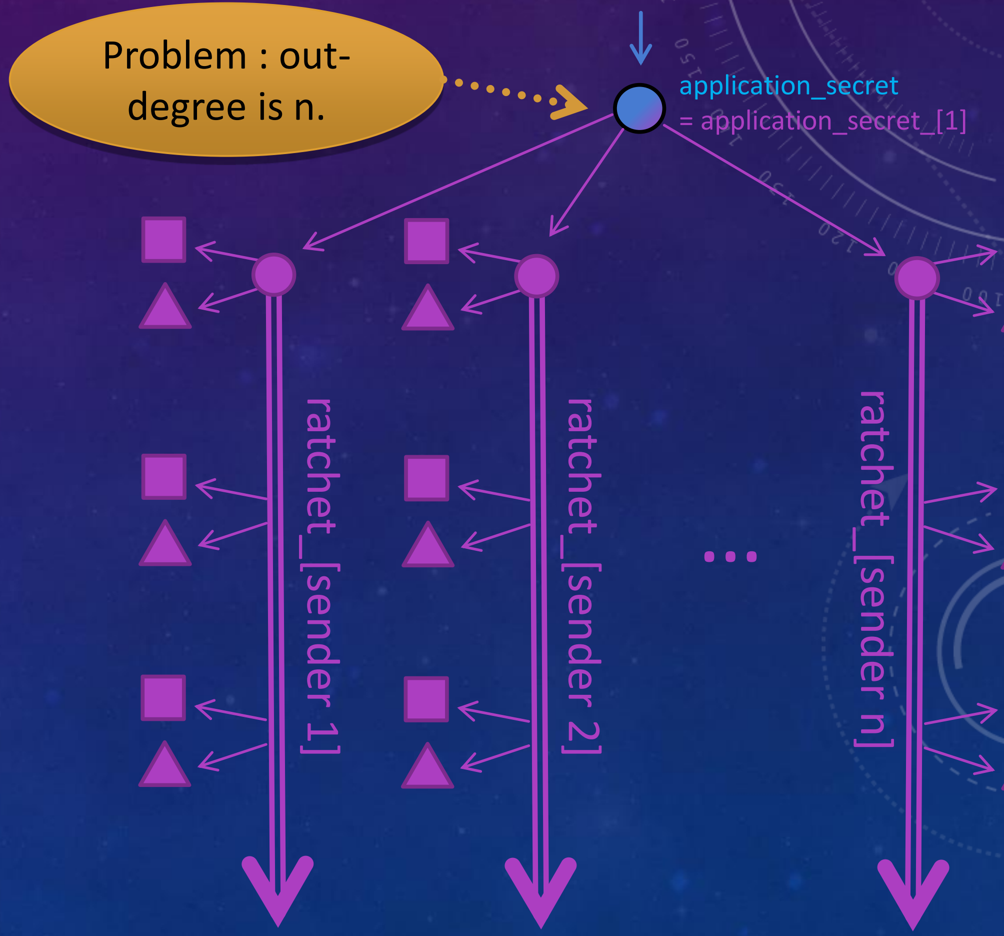
- One ratchet per group member.
- Key/Nonce j in ratchet r : encrypts j^{th} msg. of group member r.

Pro:

- Collisions between senders no longer a problem.
- Still conceptually very simple.

Con:

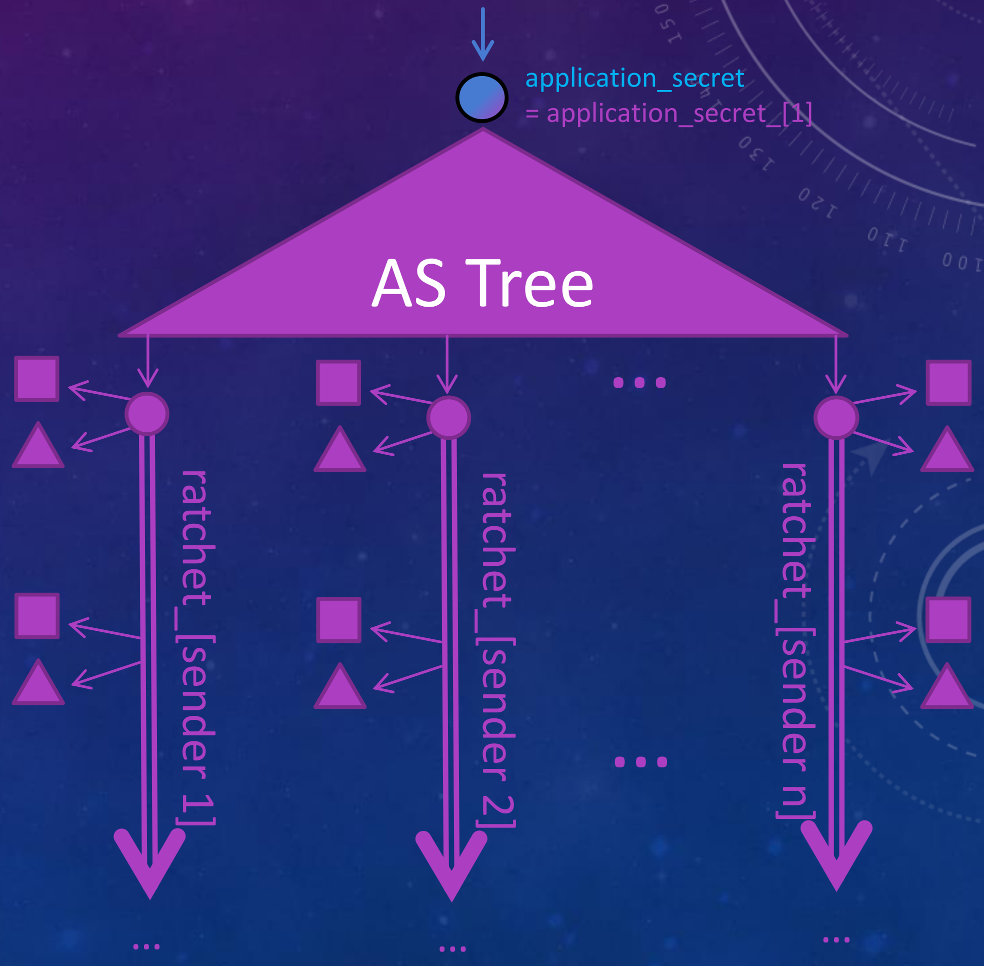
- Deletion schedule for 1^{st} msg in an epoch needs O(n) computation & memory access.
- Application key schedule state size (after 1^{st} msg) always O(n).



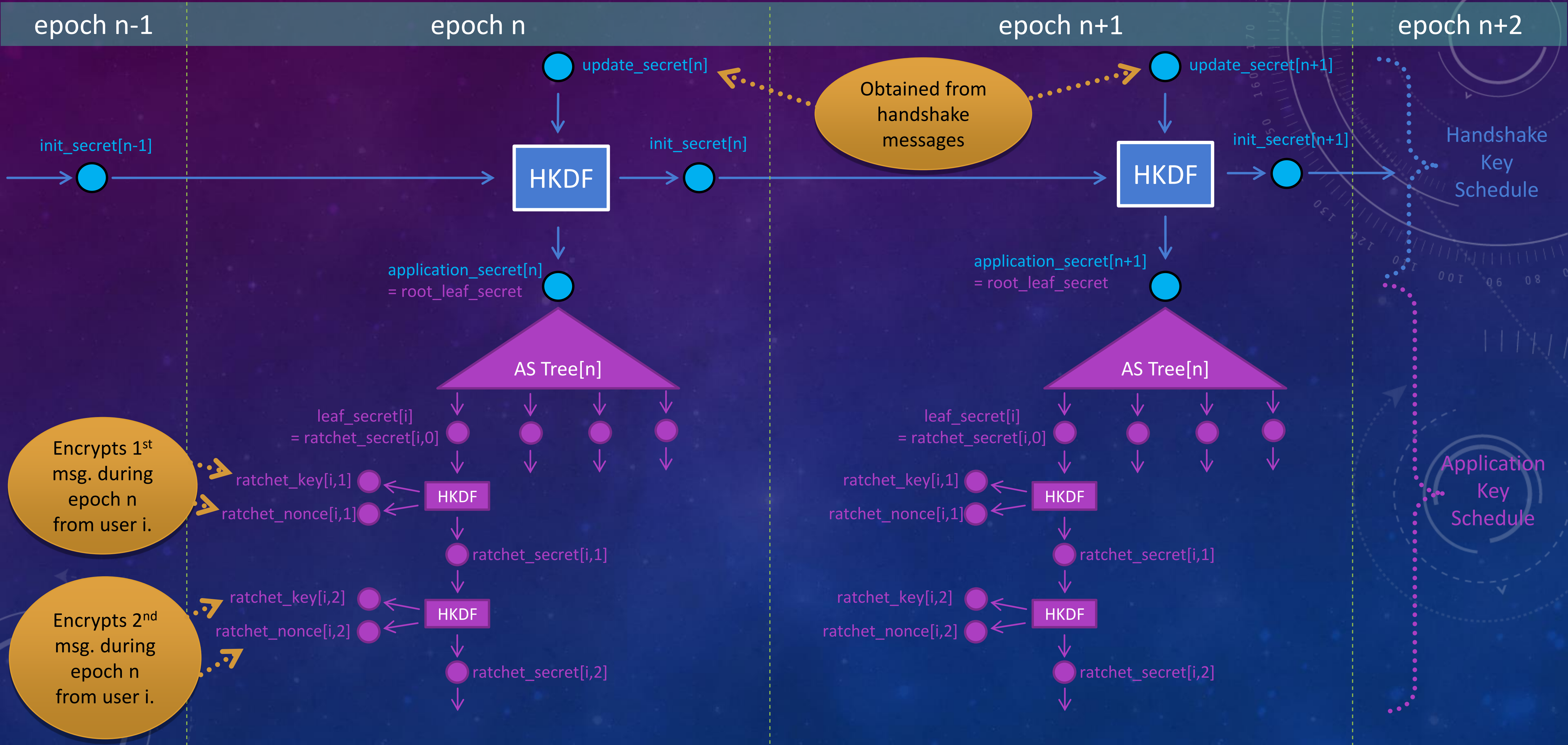
NEW APPLICATION KEY SCHEDULE: TREE-BASED

Basic Idea: Insert low out-degree “AS Tree” between root and sender ratchets.

AS Tree:
- Left Balanced Binary Tree : Identical node/edges to ratchet tree (of the current epoch).
  - Uniquely defined given group size.
  - Each group member assigned same leaf as in RT.
  - Implementation: Can piggy-back on RT data structure.
- Each node assigned secret.
  - Secret of leaf r = first secret in ratchet for sender r.



COMPLETE KEY SCHEDULE

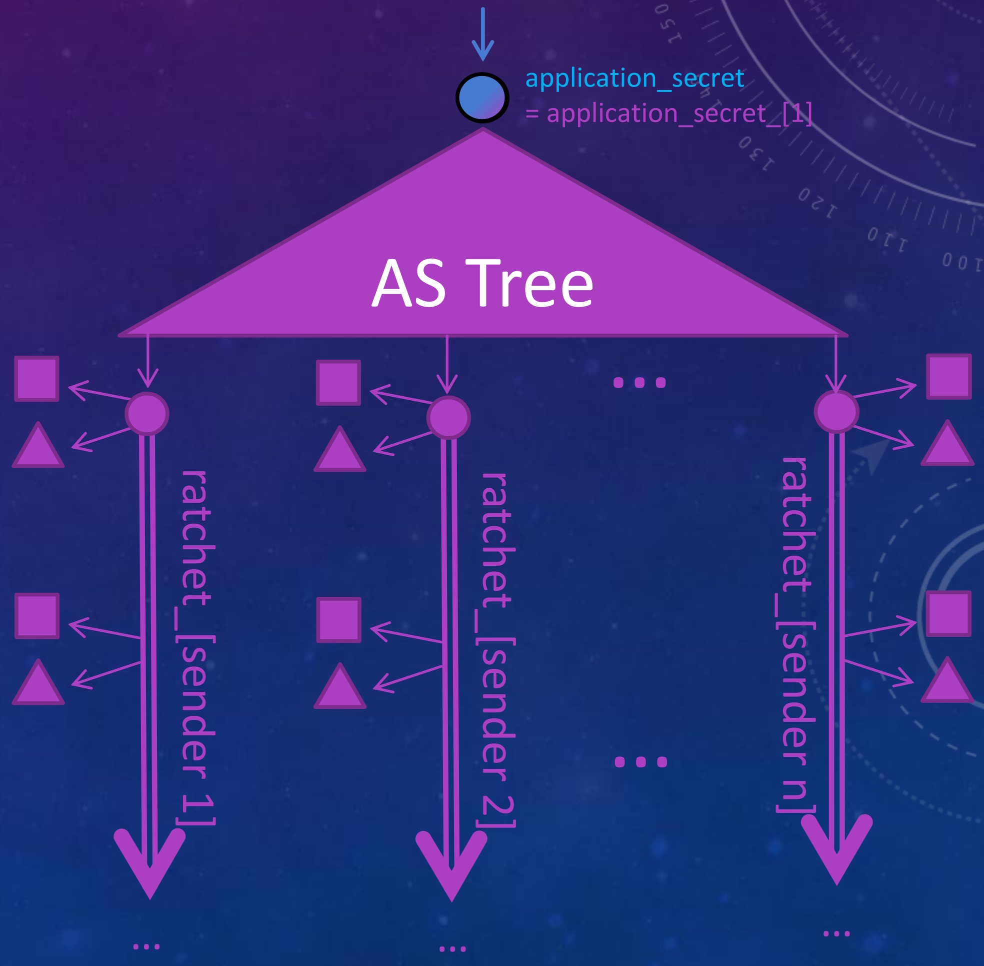


NEW APPLICATION KEY SCHEDULE: TREE-BASED

Cost of deriving any new Key/Nonce…

- Computation ≈ (consumed path length in AS Tree) × 2
  + (depth in ratchet) × 3 + 1
  ≤ log(n) × 2
  + (# of missed msgs from sender) × 3 + 1.
- Storage ≤ computation cost.

Storage after deriving any t Key/Nonce pairs
≈ (frontier length in AS Tree)
+ (# of missed msgs) × 2
≤ min(n, #active-senders × log(n))
+ (# of missed msgs from sender) × 2



DISCUSSION

- Question: Which Application Key Schedule?

- Question: How to handle failed decryption?
  1. Unwind all changes to key schedule state?

     Pro: Receiving bad message has no affect on internal state.

     Con: DOS vuln?
  2. Unwind only deletion of target leaf values?

     Pro: Don't redo computation unnecessarily

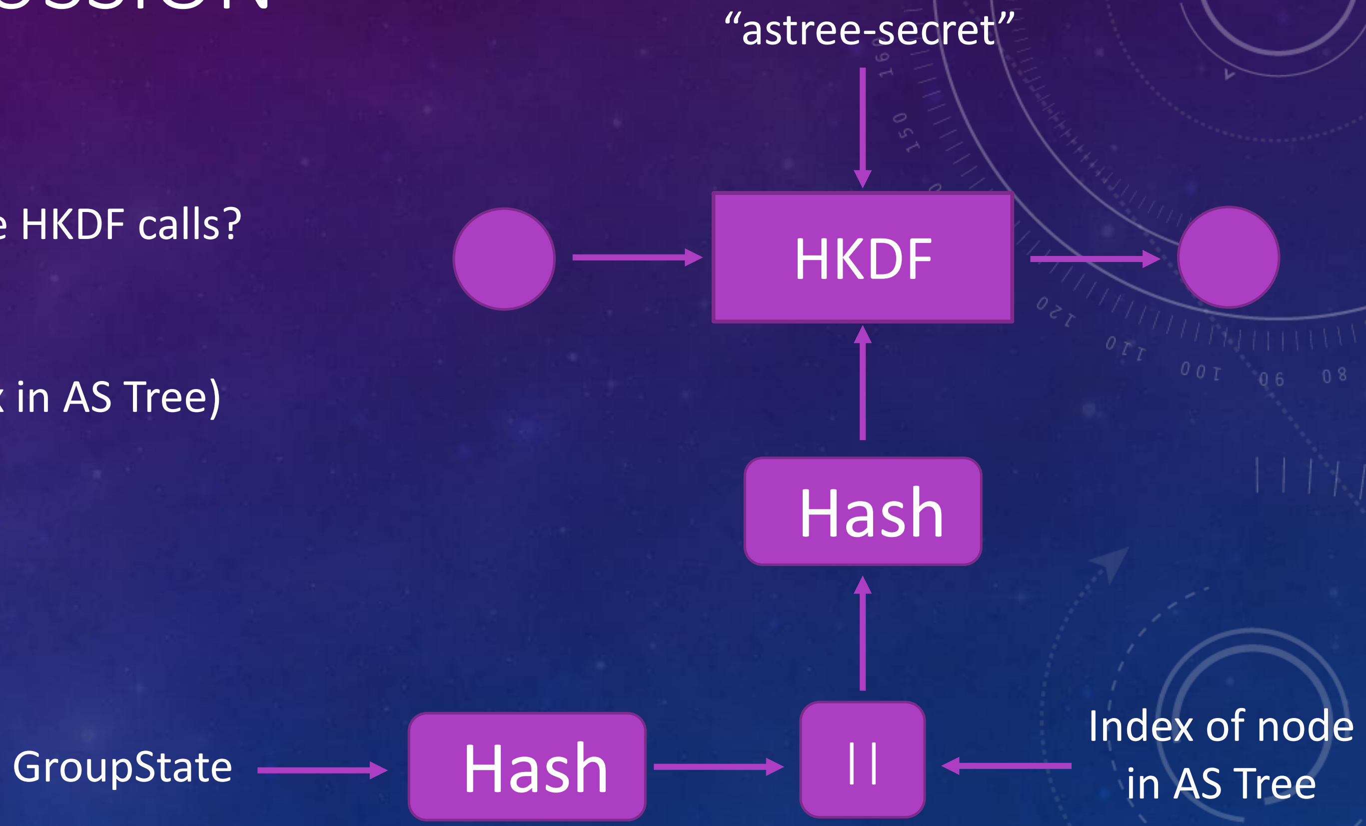
     Con: Time-attack : Leak info about past failed derivation attempts?
  3. Other solution or just unspecified?

	Complexity	Handles Sending Collisions?	Resources For 1st Key/Nonce	Resources For Other Key/Nonce
Group Ratchet	1	No	O(1)	O(1)
Sender Ratchet	2	Yes	O(n)	O(1)
Tree-Based	3	Yes	O(log(n))	O(log(n))

DISCUSSION

Question: What context & label should be included in the HKDF calls?

1. Currently for secrets in AS-Tree:
   - Label = “astree-secret”
   - Context = Hash(Hash(GroupState) || Node Index in AS Tree)

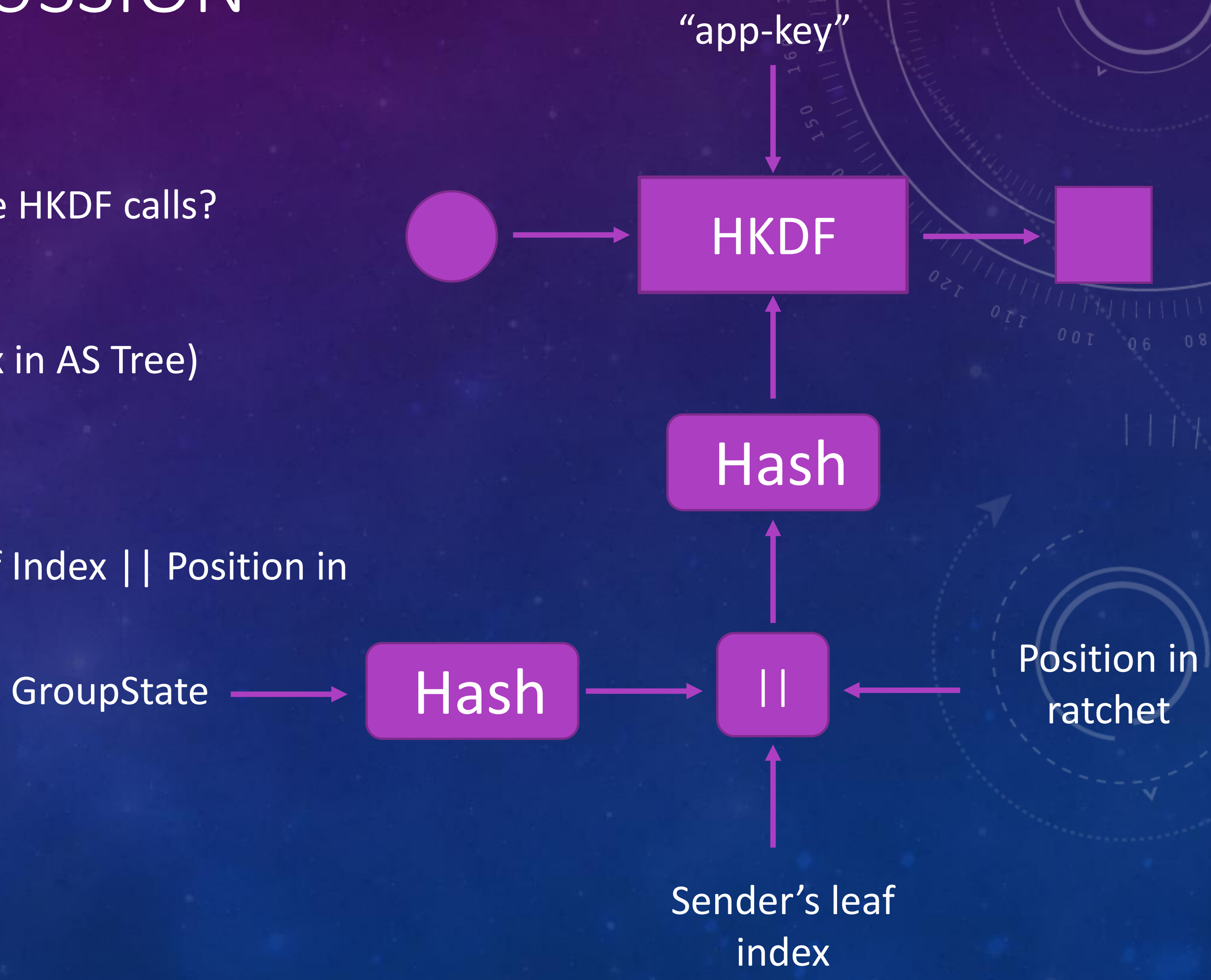


DISCUSSION

Question: What context & label should be included in the HKDF calls?

1. Currently for secrets in AS-Tree:
    - Label = “astree-secret”
    - Context = Hash(Hash(GroupState) || Node Index in AS Tree)

2. Currently for sender ratchets
    - Label = “app-nonce”, “app-key” or “app-secret”
    - Context = Hash(Hash(GroupState) || Sender leaf Index || Position in Ratchet)



DISCUSSION

Question: What context & label should be included in the HKDF calls?

1. Currently for secrets in AS-Tree:
    - Label = “astree-secret”
    - Context = Hash(Hash(GroupState) || Node Index in AS Tree)

2. Currently for sender ratchets
    - Label = “app-nonce”, “app-key” or “app-secret”
    - Context = Hash(Hash(GroupState) || Sender leaf Index || Position in Ratchet)

Con:

- Possibly redundant?
- Requires extra Hash per KDF call (but so would any non-empty other context)

Pro:

- Defense in Depth.
- Not too expensive. (E.g. Hash(GroupState) already needed elsewhere.)