draft-ietf-mls-protocol

IETF 105

Changes in -06 and -07

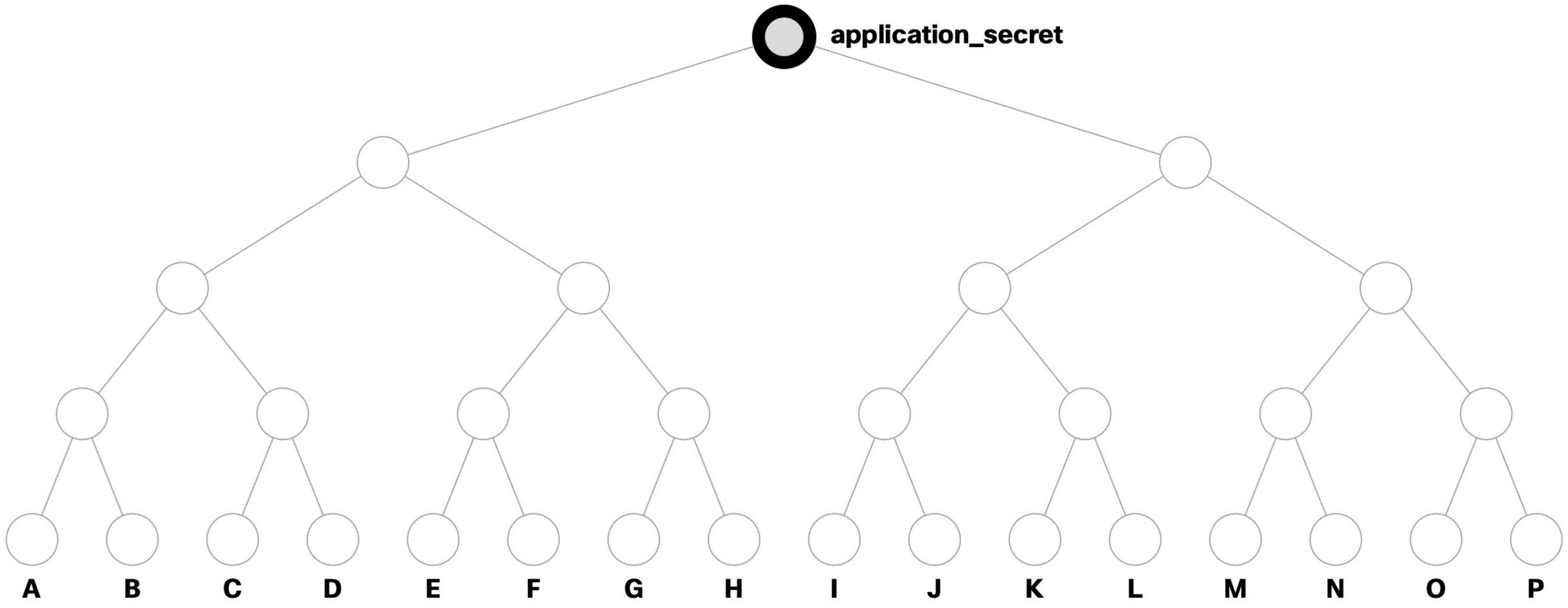
draft-06 - Bugfixes & Interim Feedback

- GroupState -> GroupContext, UserInitKey -> ClientInitKey
- Fixes to the key schedule and common framing introduced in draft-05
- Reorder blanking and update in the Remove operation

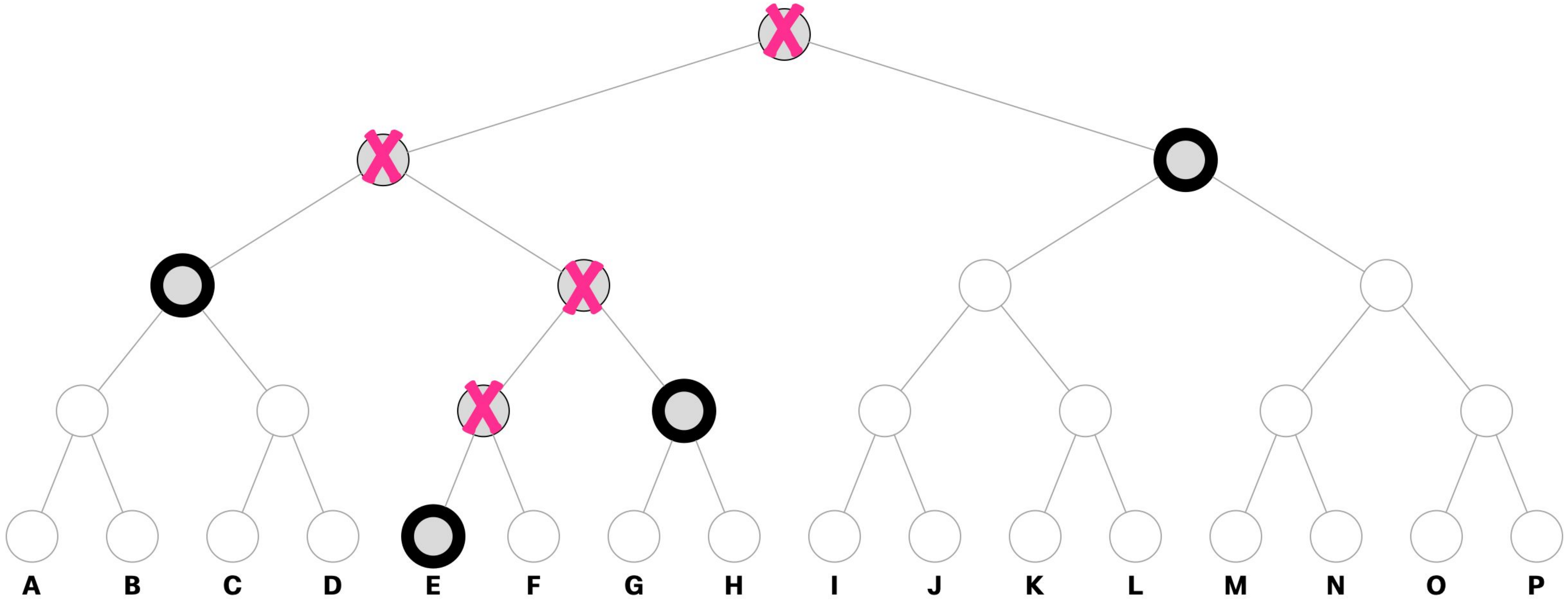
draft-07 - Optimizations

- Expand transcript to cover more of the handshake
- Enable new joiners to decrypt the Add
- Tree-based application key derivation
- Consumption rules
- Init message

Application Secret Tree (new epoch)



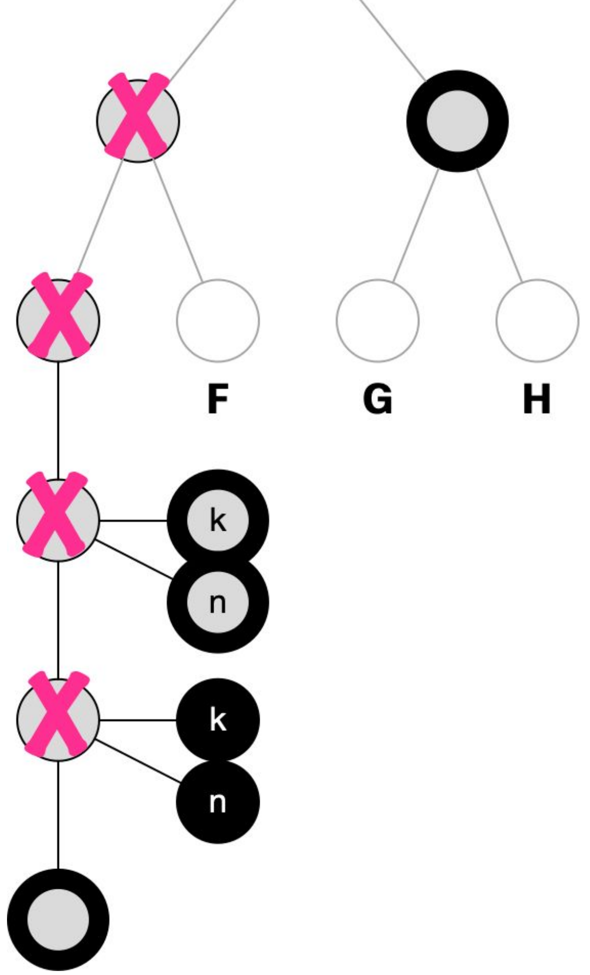
Message from E



Consuming Secrets

A secret S is consumed if:

- It is used to decrypt a message
- A secret derived from S has been consumed



Next couple of targets

draft-08 for interim in September 2019

- Incorporate feedback from this meeting
- Hopefully start getting implementations aligned

draft-09 for IETF 106 in November 2019

Approaching WGLC toward end of year?

Laziness

Problems

Group secret only needs to be updated if there’s messaging

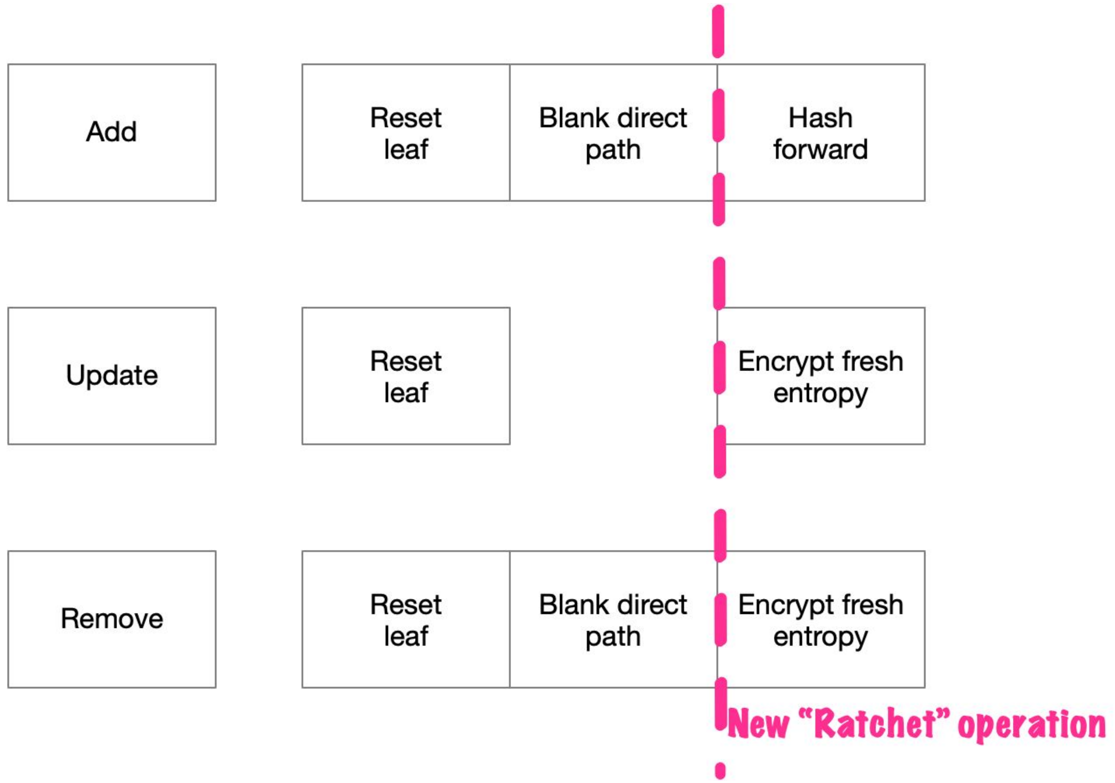
Current operations update group secret on every change to the group, whether or not there’s any messaging happening

All operations need to be originated by a member of the group; a third party can’t initiate an add or remove

Recombination

Add	Reset leaf	Blank direct path	Hash forward
Update	Reset leaf		Encrypt fresh entropy
Remove	Reset leaf	Blank direct path	Encrypt fresh entropy

Recombination



[[Illustration]]

Protocol Impact

Split the secret-update part of the current operations into a new Ratchet operation

If any Add/Update/Remove have been sent, then a sender MUST send a Ratchet before sending an application message

Trade-Offs

On the one hand:

- Add / Update / Remove are constant time
- Ratchet is more expensive, due to blanking (see later presentation, though)
- Add* / Update / Remove can be sent by any authorized party

On the other hand:

- Risk of bugs in Ratchet timing

Server Add

Server-Initiated Add

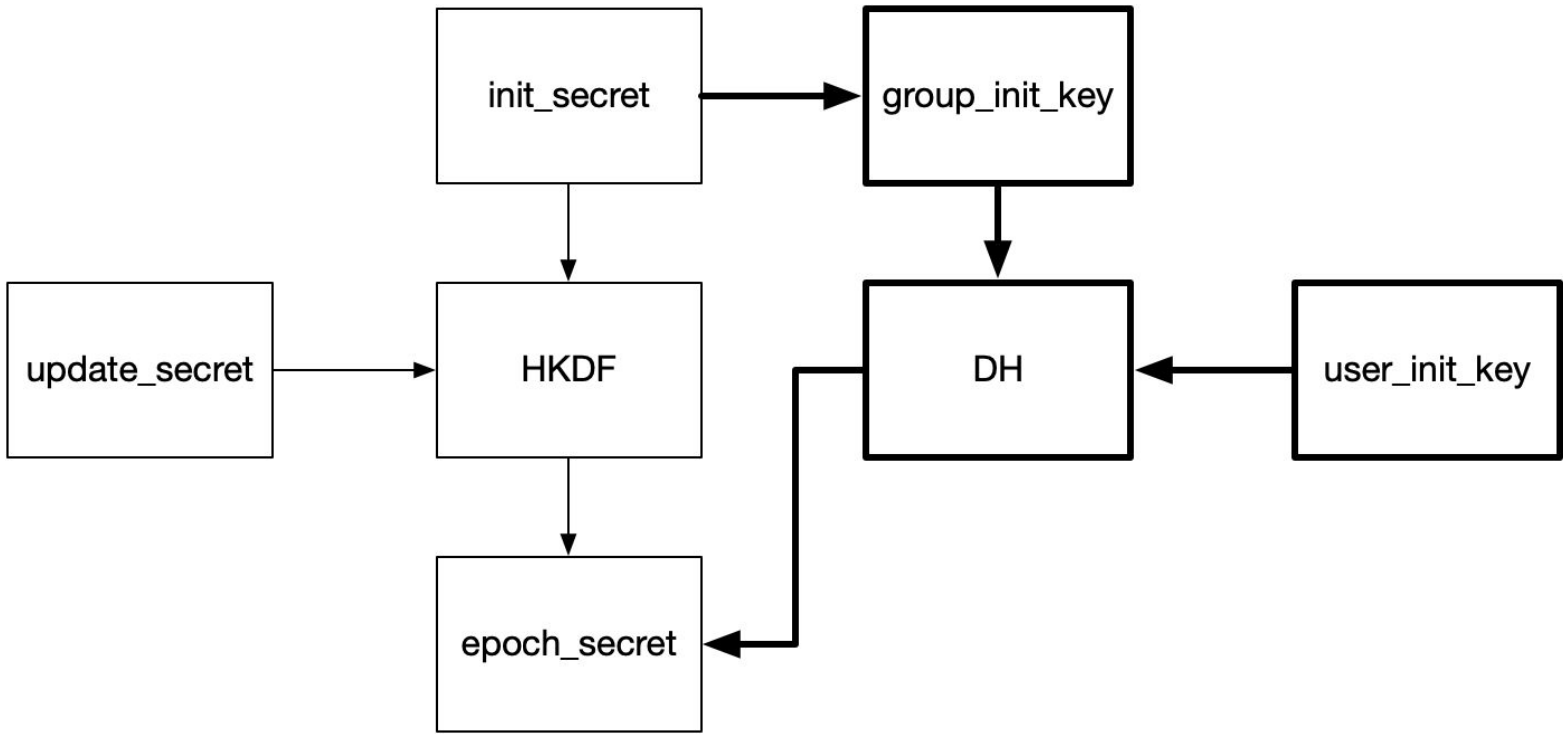
In lazy framing, the server can synthesize an Add, but not a Welcome

… because it doesn’t have the init secret

Bring back GroupAdd: Asymmetric ratchet on add instead of KDF

- Group publishes a public key whose private key is held by members
- Welcome has public key instead of init secret
- New epoch secret formed via DH between group and user key pairs

Asymmetric Init Secret



Protocol Impact

```
struct {
    ProtocolVersion version;
    opaque group_id<0..255>;
    uint32 epoch;
    optional<RatchetNode> tree<1..2^32-1>;
    opaque interim_transcript_hash<0..255>;
    HPKEPublicKey init_pub;
    optional<KeyAndNonce> add_key_nonce;
} WelcomeInfo;
```

Trade-Offs

On the one hand:

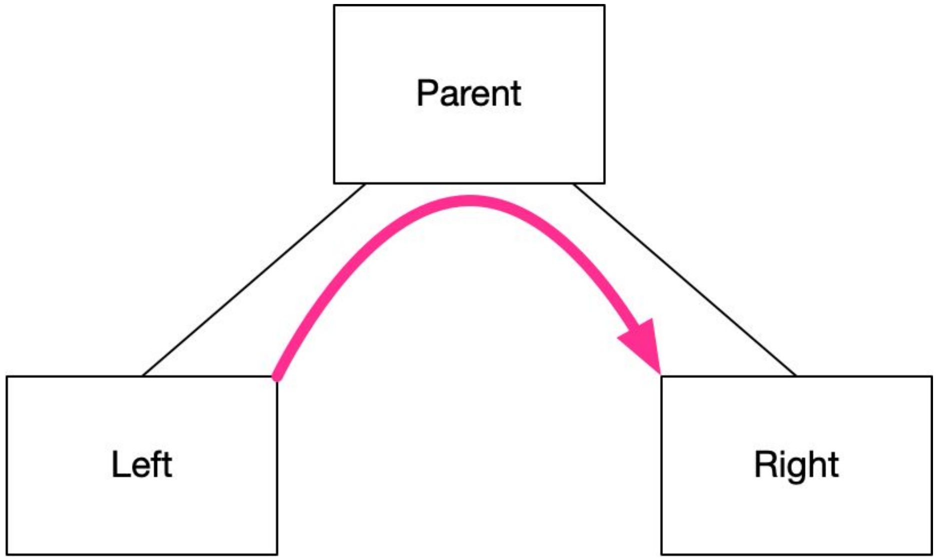
- Unified logic for user- or group-initiated Add
- Any authorized party can synthesize a Welcome
- No more passing around secrets

On the other hand:

- Possibly complicated authorization story
- Locks us into DH-like constructions (?)

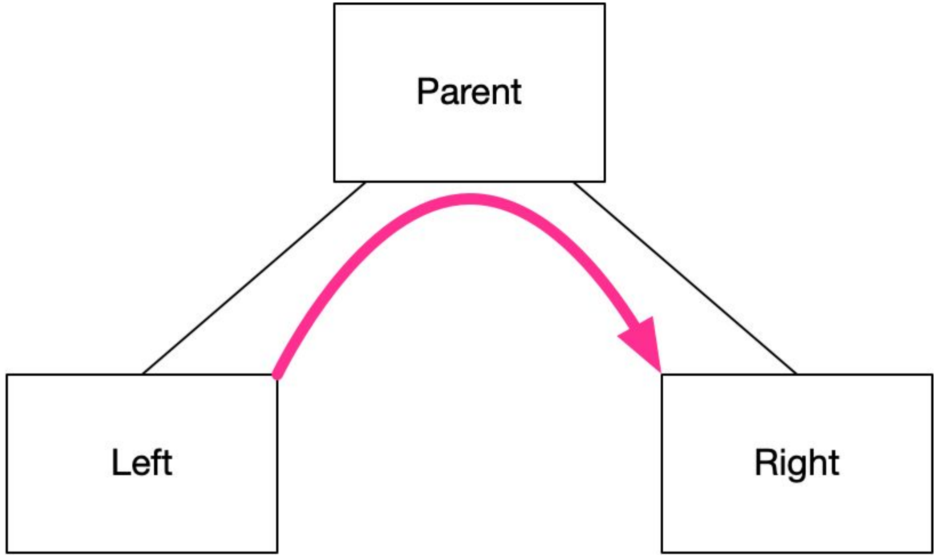
Which HPKE?

Encrypting to sibling



```
E(pkR=Right, psk=None, pkI=None, info=None; path_secret_[Parent])
```

More binding?



```
E(pkR=Right, psk=something, pkI=Left, info=something; path_secret_[Parent])
```

Trade-Offs

On the one hand:

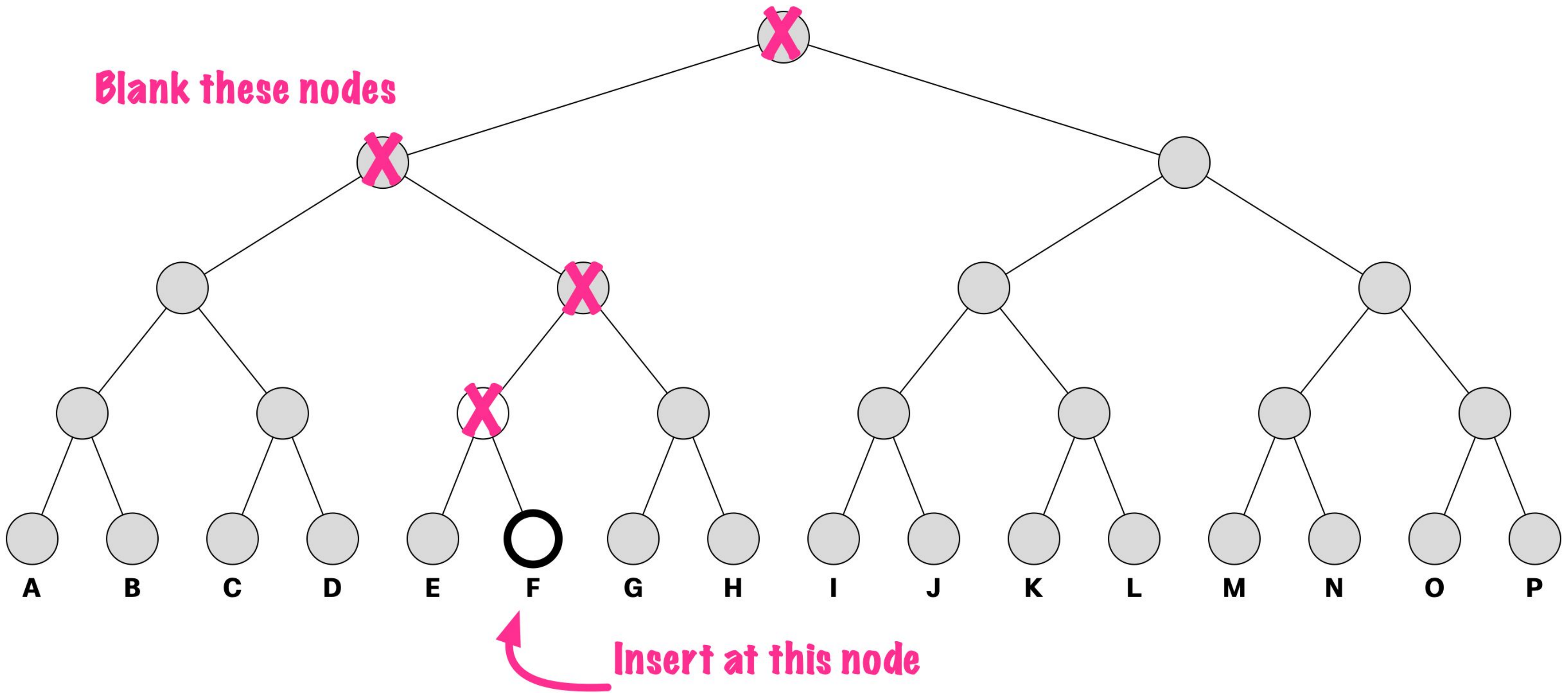
- Ciphertexts are bound to a specific group, epoch, and tree position
- Populating PSK or info inputs is has no crypto cost

On the other hand:

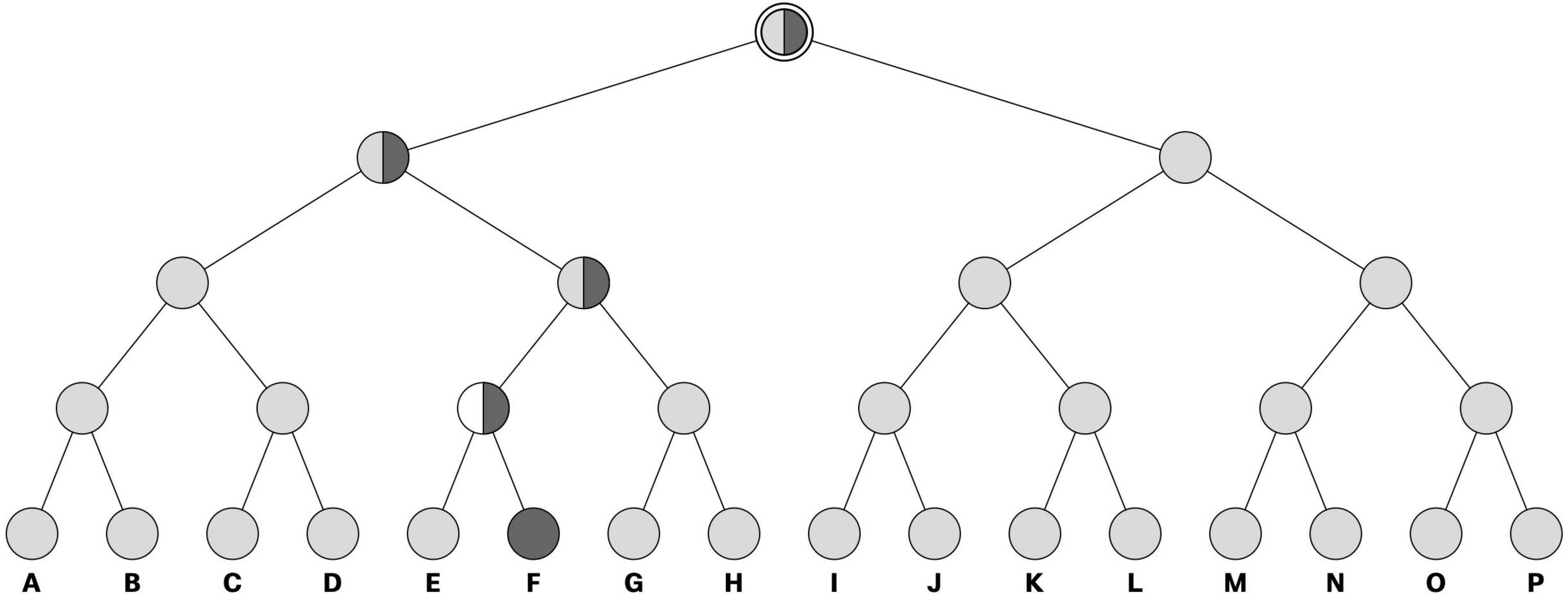
- PSK is yet another group secret to derive
- Asymmetric authentication requires an extra DH operation…
- … and constrains us to use DH / AuthKEM constructions

Non-Destructive Add

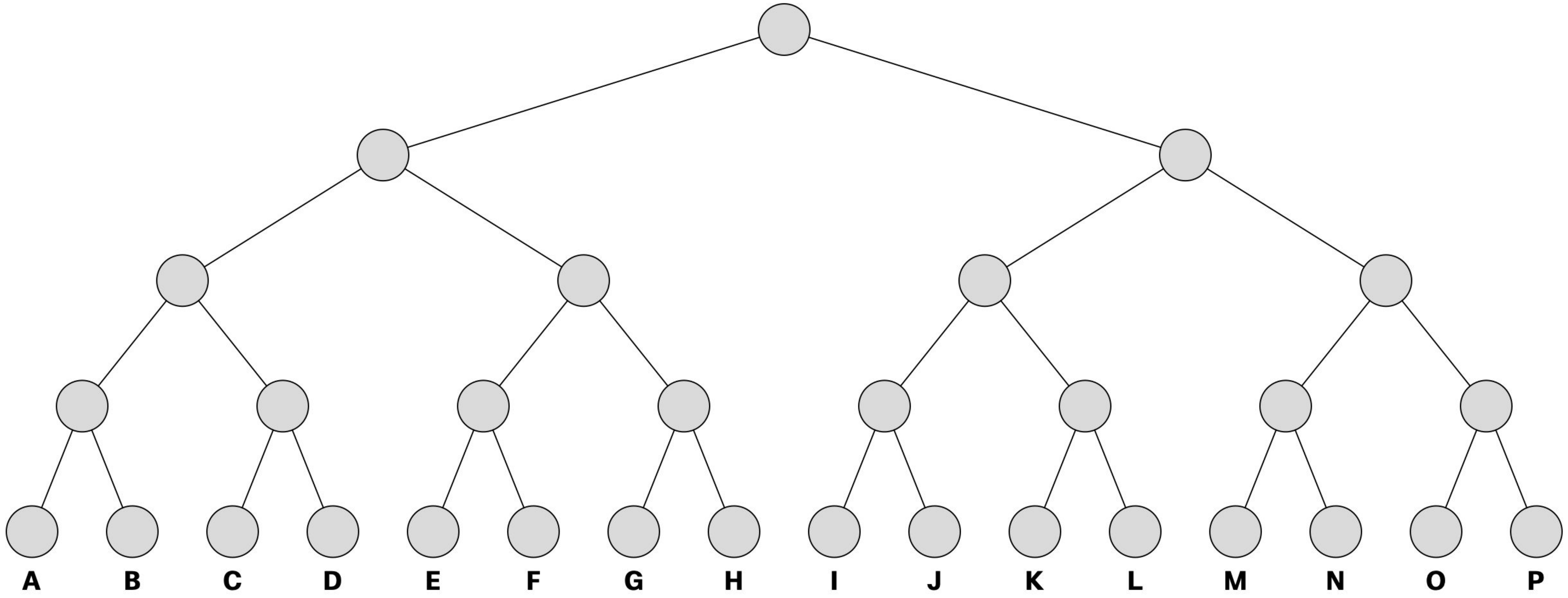
Recall: Add = Emplace + Blank dirpath



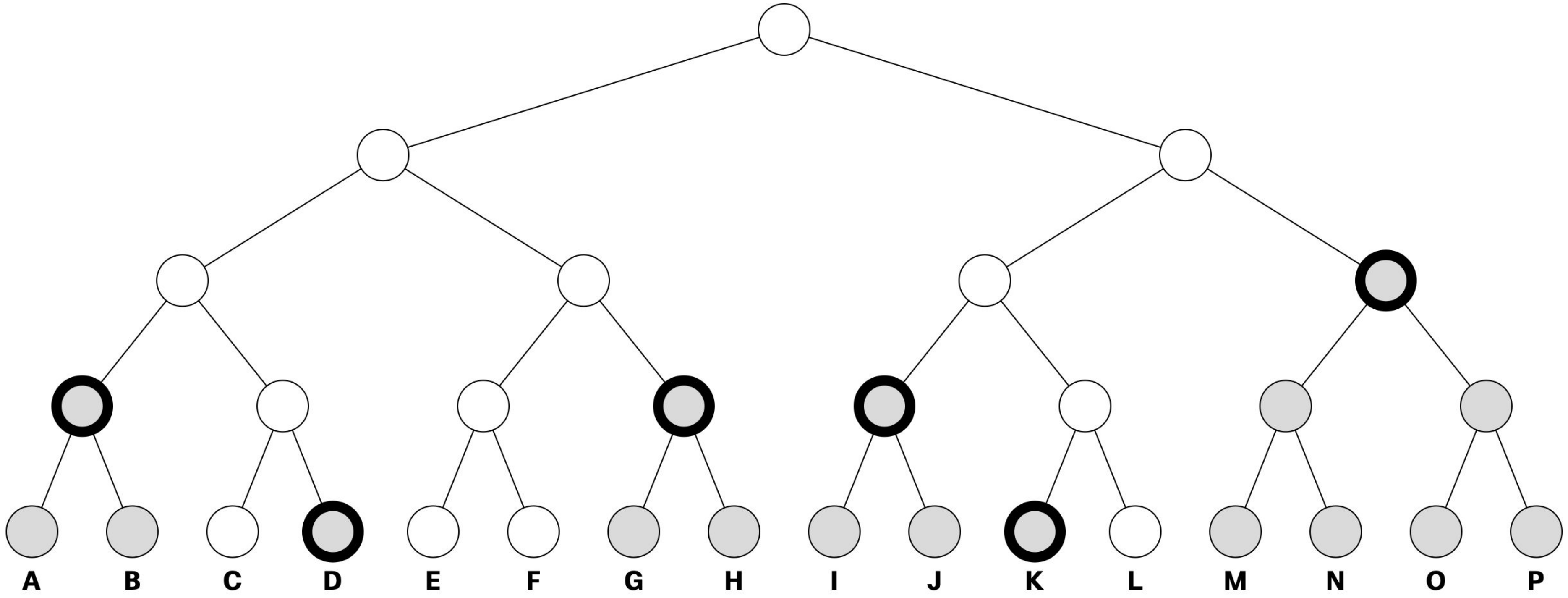
Don't blank, add leaf keys



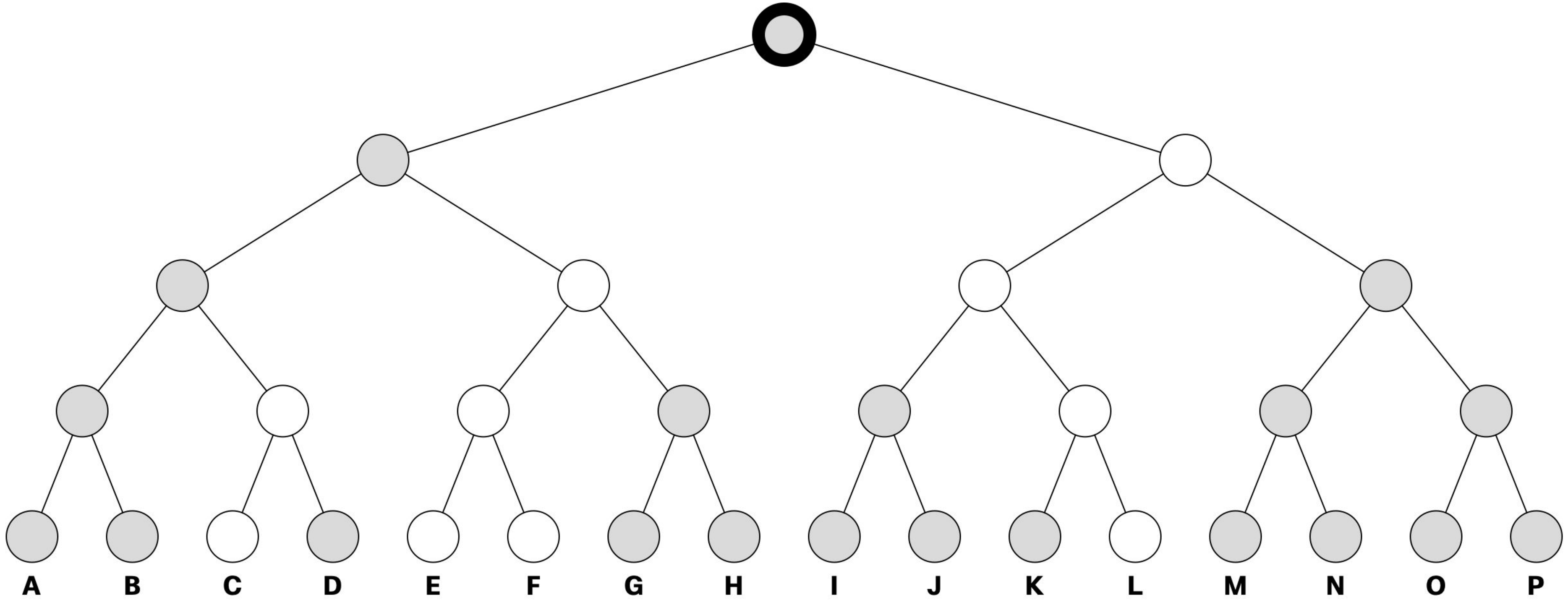
Suppose you've got a full subtree...



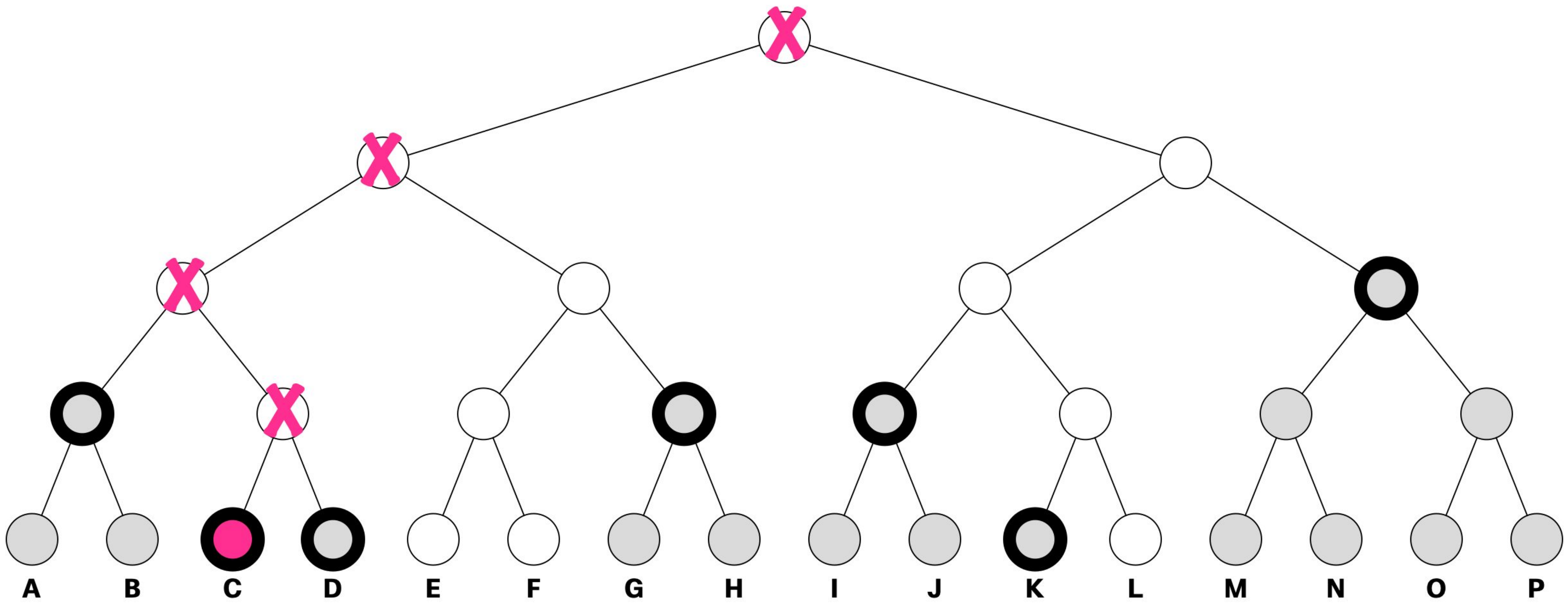
Remove C, E, F, L => 6 heads



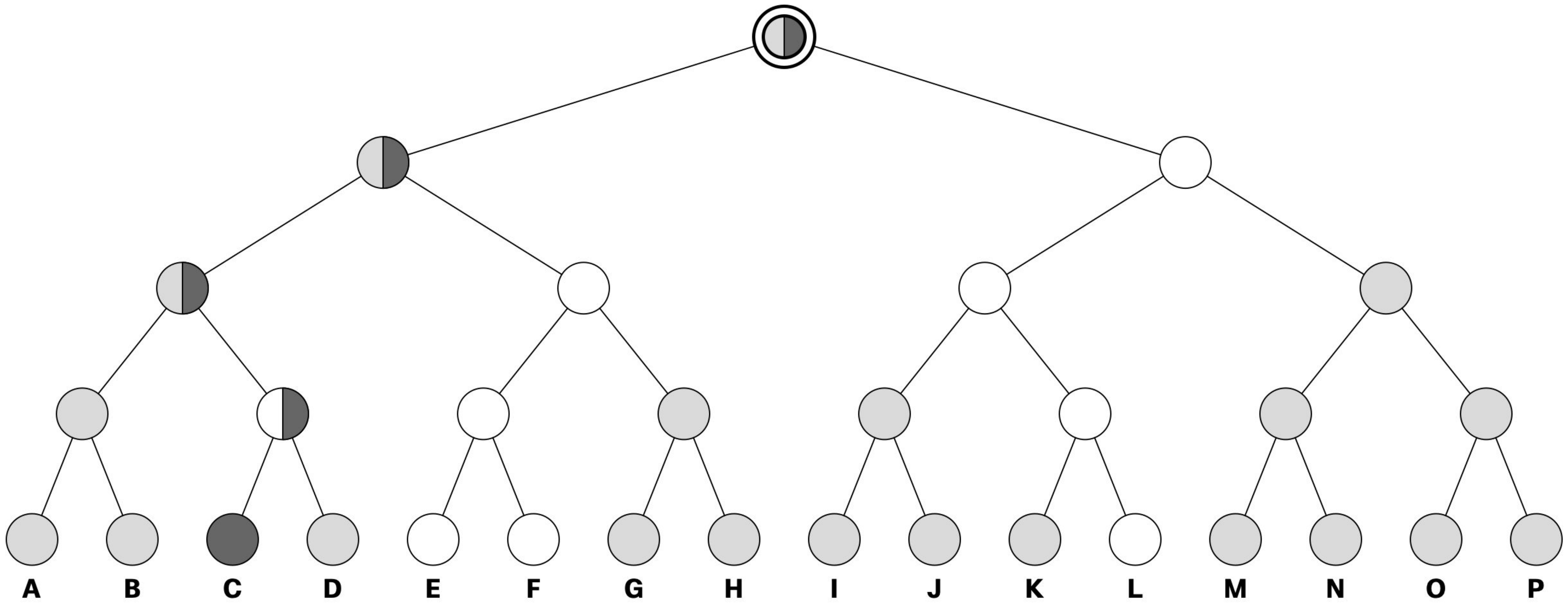
Update A => 1 head



Add @ C (destructive) => 7 heads



Add @ C (non-destructive) => 2 heads



Protocol Impact

Update algorithms to track “unmerged leaves”

Update resolution algorithm to encrypt to unmerged leaves as well as intermediate nodes

```
struct {
  HPKEPublicKey public_key;
  optional<Credential> credential;
  uint32 unmerged_leaves<0..2^16-1>;
} RatchetNode;
```

Trade-Offs

On the one hand:

- Less degradation in the tree due to Add => fewer public-key operations

On the other hand:

- More record-keeping and complexity

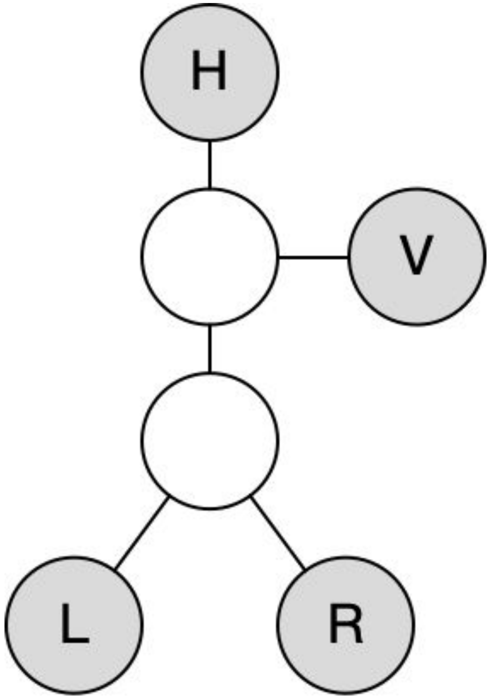
Tree Signing

The Lying Adder

When someone adds you to a group, they can lie about the content of the tree

Even if the leaves are correct, the intermediate keys might not meet the tree invariant

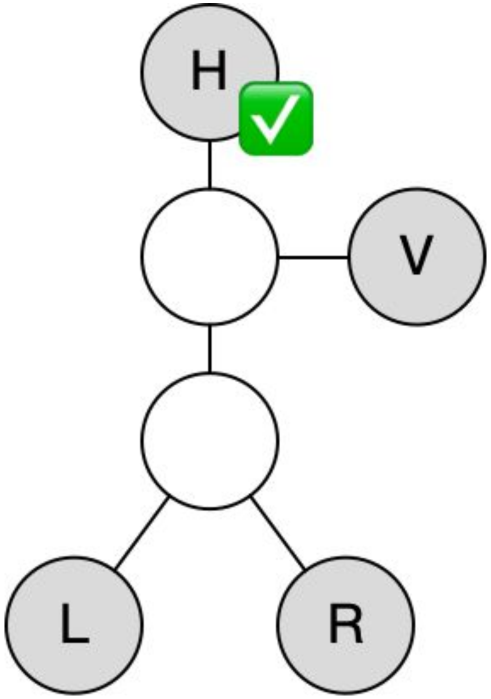
Recall: Tree Hashes



Each node in the ratchet tree has an associated hash, computed from its children and value

We use this to confirm agreement on the tree

Sign the Tree Hash

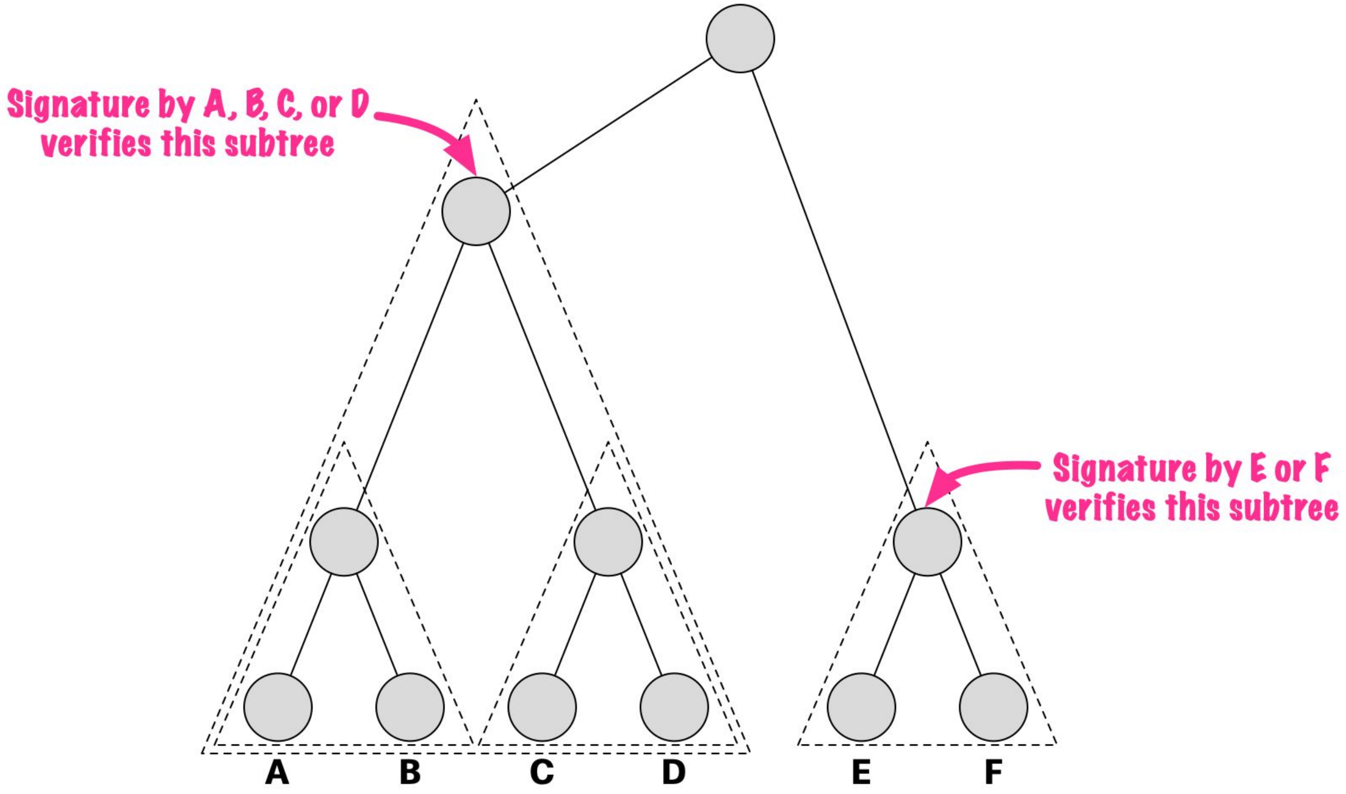


Whenever a member changes a node (e.g., with an Update), they sign the node's new hash

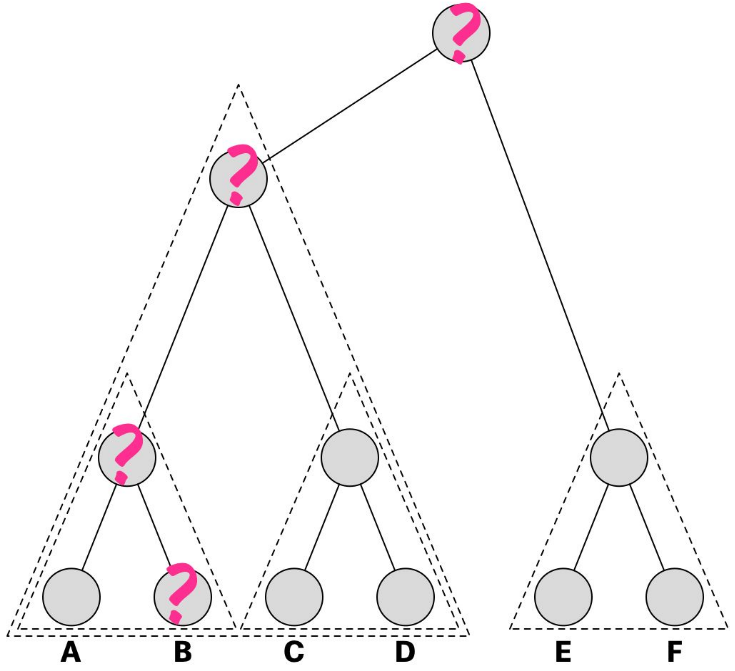
Thus, a node can only tamper with intermediate nodes in its direct path

Full security impact is a little hard to characterize succinctly

More Rigidity in the Tree


Signature by A, B, C, or D verifies this subtree
Signature by E or F verifies this subtree
A
B
C
D
E
F

Containing the Lying Adder



Protocol Impact

Need to transmit these signatures in Welcome / Add / Update / Remove

32 -> 32 + 64 B/node

```
struct {
  HPKEPublicKey public_key;
  optional<Credential> credential;
  uint32 signer_id;
  opaque signature<0..2^16-1>;
} RatchetNode;

struct {
  HPKEPublicKey public_key;
  HPKECiphertext encrypted_path_secret<0..2^16-1>;
  uint32 signer_id;
  opaque signature<0..2^16-1>;
} DirectPathNode;
```

Trade-Offs

On the one hand:

- Some protection against adder providing bad tree information
- Adder can only lie about nodes in its direct path

On the other hand:

- ~3x increase in size of update