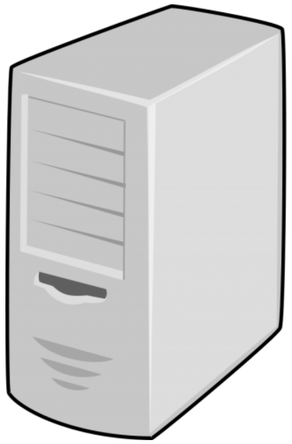
CS 350S: Privacy-Preserving Systems

Key Transparency

Outline

1. CONIKS lecture
2. Guest lecture from Kevin Lewi (Meta)

Motivation: End-to-end encrypted messaging



Messaging provider

$\mathsf{Enc}(\mathsf{pk}_{\mathsf{Bob}}, m)$

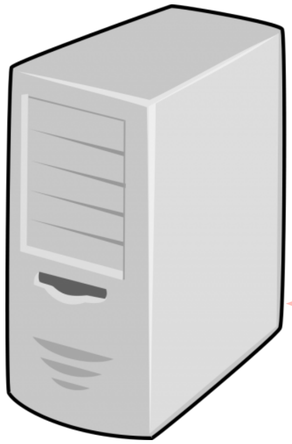
$\mathsf{Enc}(\mathsf{pk}_{\mathsf{Bob}}, m)$

Challenge: How does Alice get $\mathsf{pk}_{\mathsf{Bob}}$?

Secret key $\mathsf{sk}_{\mathsf{Bob}}$

Challenge: Man-in-the-middle attacks

Secret key $\mathbf{sk}_{\mathbf{evil}}$



Messaging provider

A compromised messaging provider can provide the wrong public key and break confidentiality

Bob: $\mathbf{pk}_{\mathbf{evil}}$

$\mathbf{Enc}(\mathbf{pk}_{\mathbf{evil}}, m)$

Secret key $\mathbf{sk}_{\mathbf{Bob}}$

Tool: Transparency logs

- Idea: Use a transparency log to map usernames to public keys
- Goal: *Detect* when the wrong public key is advertised for a user
  - Not to prevent the wrong public key from being advertised
- Why is detection enough?
  - A user who catches misbehavior can alert the other users (whistleblowing), leading to loss of trust in the provider
  - Providers generally care about their reputation

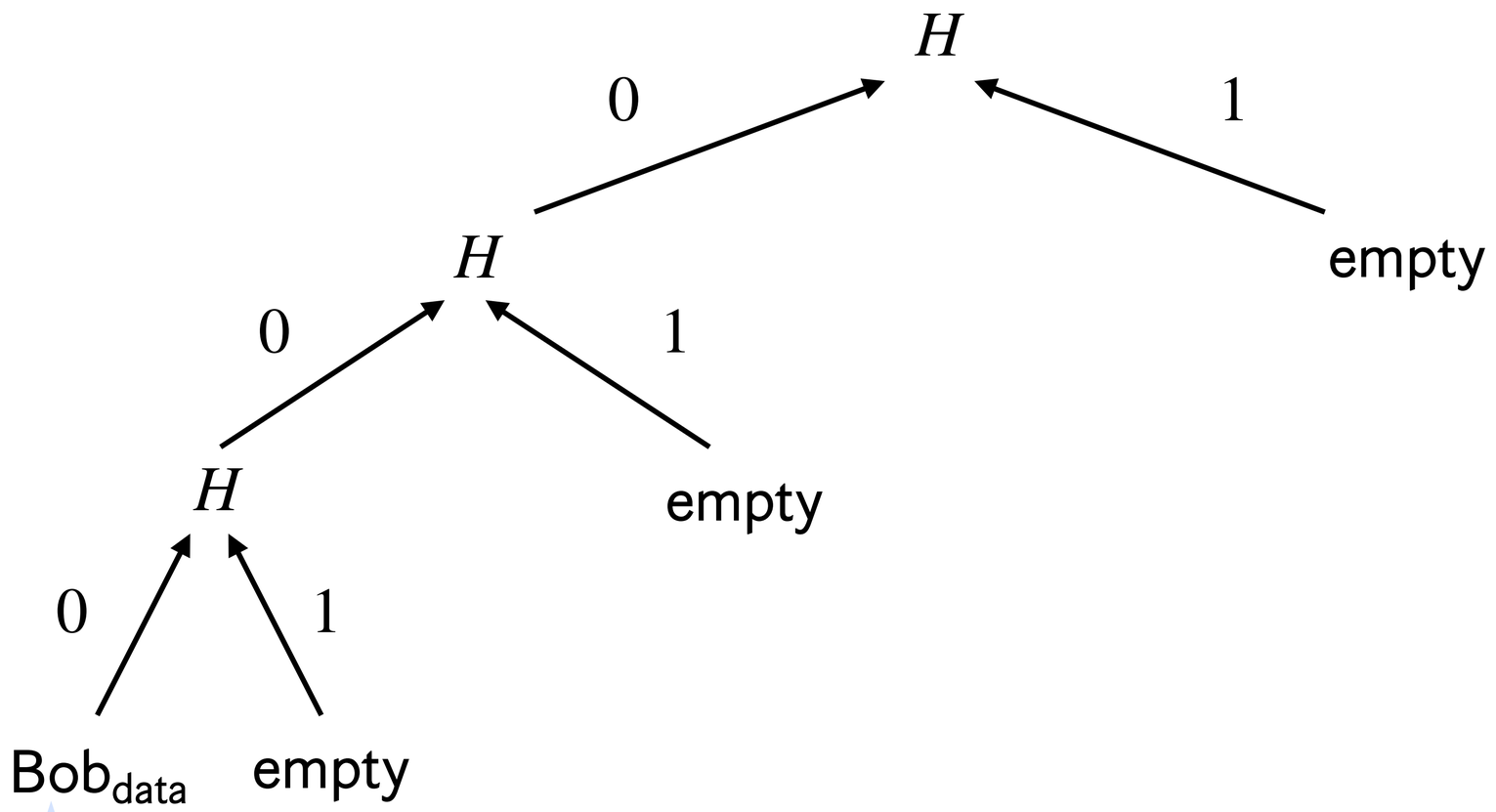
Challenges in using transparency logs for user keys

- Efficient monitoring
  - Certificate transparency: monitor by scanning contents of entire log
  - Key transparency: clients don’t have the resources to check the entire log
- Privacy
  - Certificate transparency: all certificates are public
  - Key transparency: users should not be able to see all other application users

Security and privacy properties

- **Non-equivocation:** If an identity provider equivocates (i.e., presents different views to different users), the diverging views must be maintained forever, or there will be publicly verifiable evidence of equivocation
- **Detecting malicious keys:** If an identity provider changes the public key for a user, the user's client will be able to detect this.
- **Hide the identity of other users:** Users cannot learn information about what other usernames are in the system when checking key-binding consistency. (Users can still learn about other users by looking up public keys.)
- **Hide the number of users:** Users can only learn some upper bound on the number of users, where this upper bound is dependent on the number of "dummy entries" inserted by the provider.

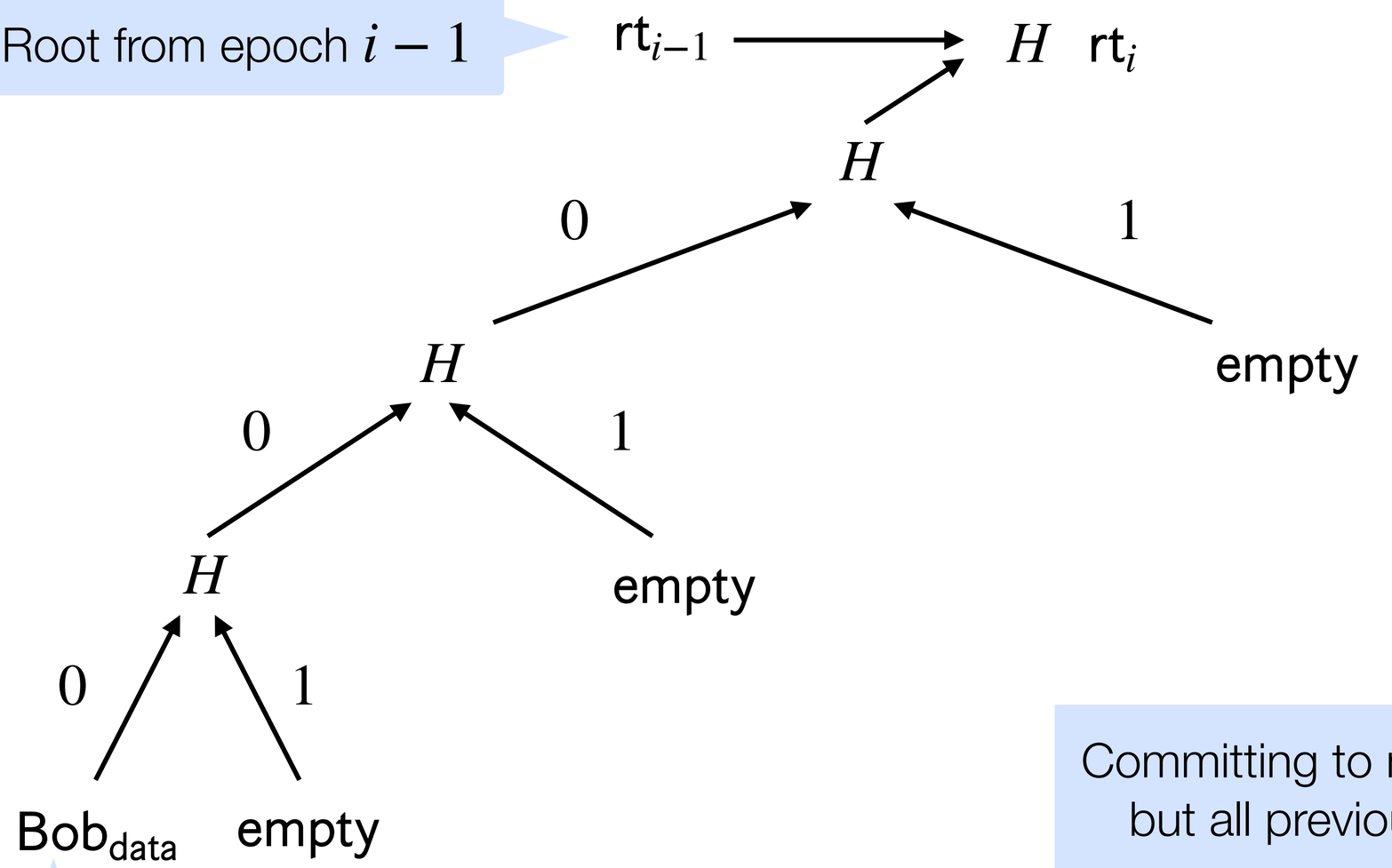
Merkle prefix tree



Bob's index is 000

Merkle prefix tree with hash chain

Root from epoch $i-1$

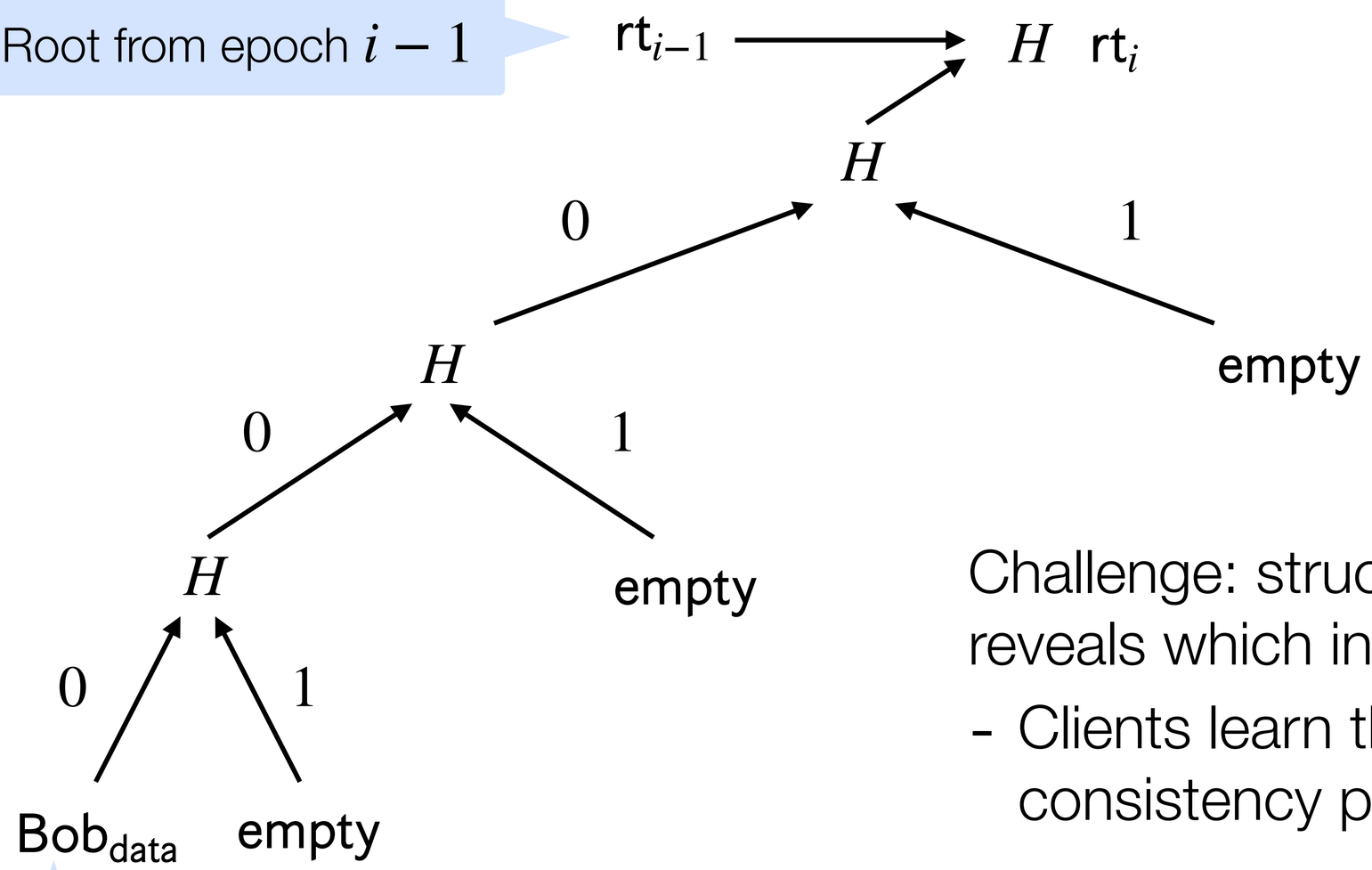


Committing to not only current state, but all previous versions of state

Bob's index is 000

User privacy

Root from epoch $i-1$



Bob's index is 000

Challenge: structure of tree reveals which indexes are in tree

- Clients learn the structure via consistency proofs

Privacy for usernames

Verifiable unpredictable function (VUF):

- Use the secret key to generate an unpredictable value v for input x
- Use the public key to verify that the unpredictable value v is correct for input x
- Can implement with deterministic signature scheme

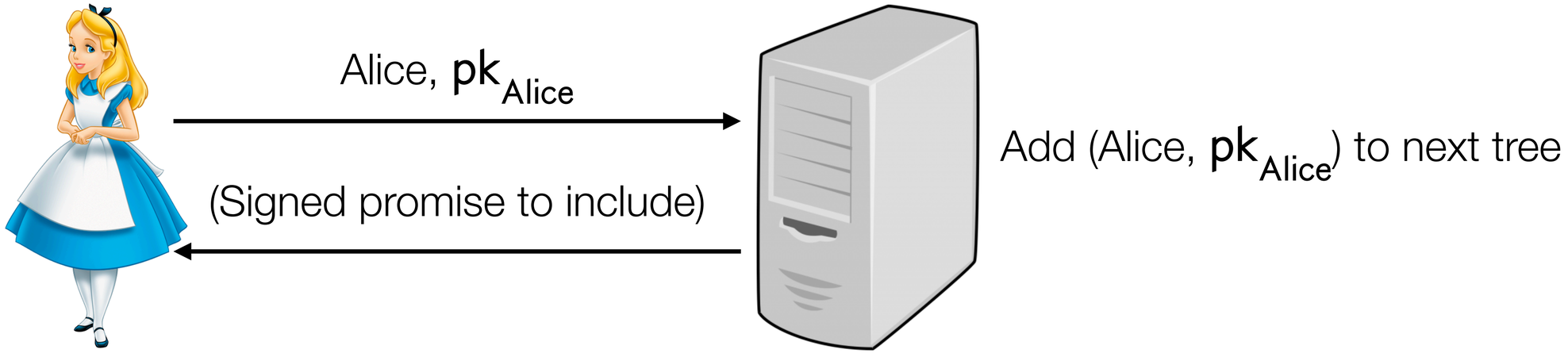
Provider uses VUF secret key to generate an unpredictable index for Bob.

Clients can use VUF public key to verify that the identity provider outputs the right index for a client.

What goes wrong if not deterministic? Public key may be in multiple locations

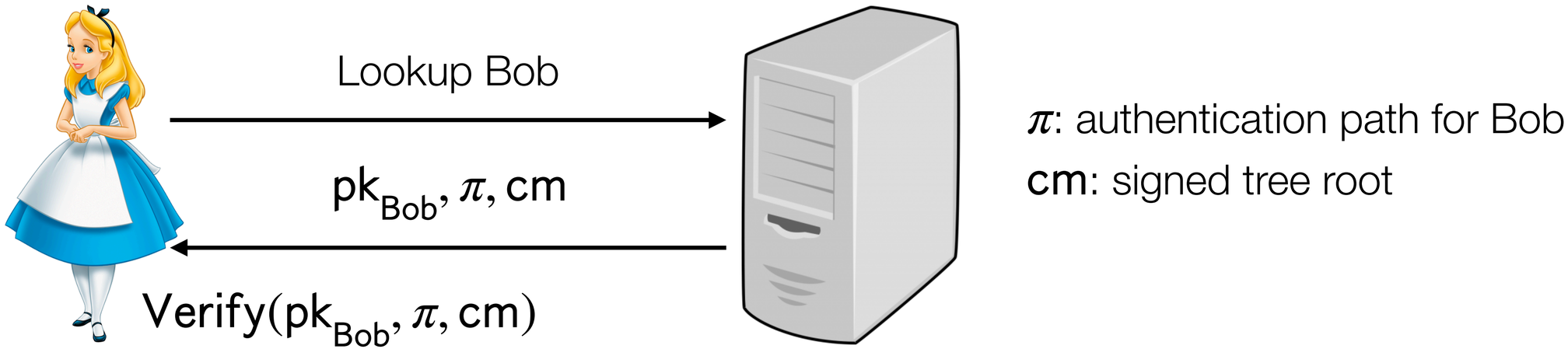
Registering a key

- Client sends public key and username to the provider
- Provider adds (username, public key) to list of updates to include in next tree
- Optional: the provider sends back a signed promise to include the binding in the next epoch (when new tree is released)
  - Makes it possible to use new keys before next epoch
  - In next epoch, client checks that the server kept its promise



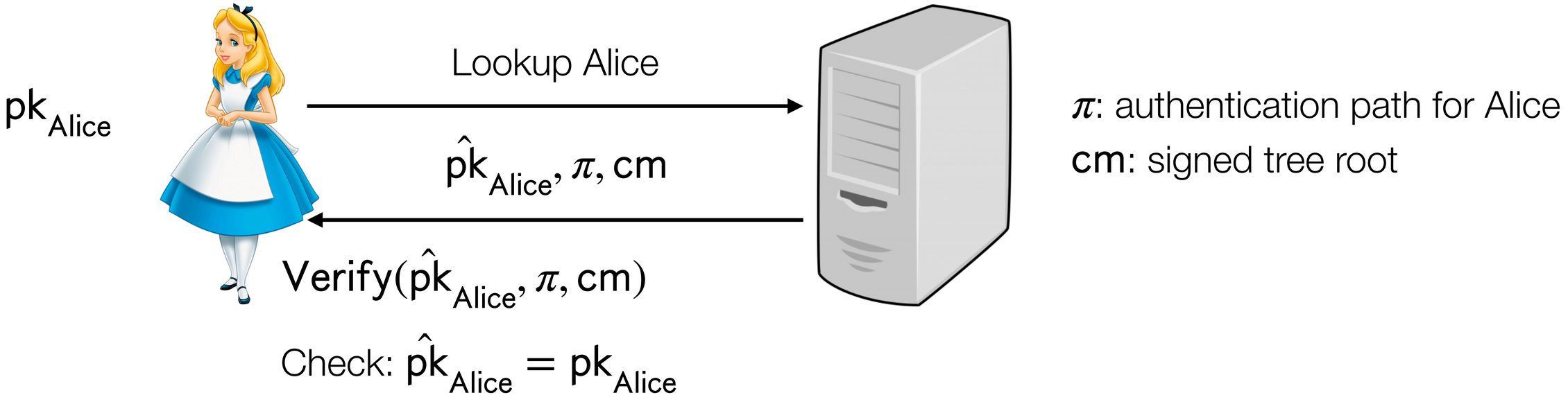
Looking up a key

- Client sends username to provider
- Provider responds with public key, authentication path, and signed tree root
- Client verifies the authentication path: checks Merkle proof and that index was computed correctly



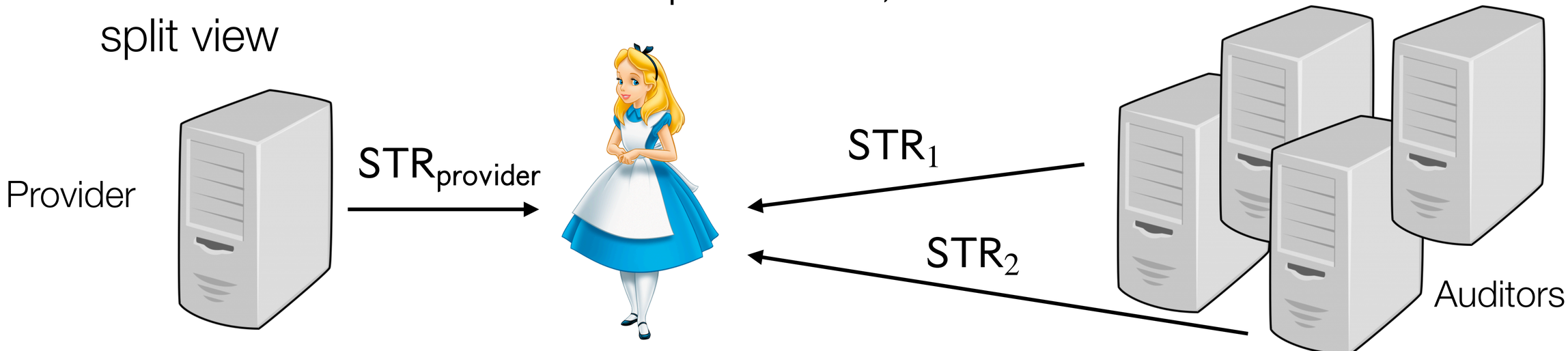
Monitoring a key binding

- Goal: check that Alice's key binding is correct
- Alice performs the same lookup procedure, but for her own key
- She also checks that the returned key matches the expected public key



Auditing for consistency

- Goal: Ensure that clients are seeing the same, linear history of key bindings
- Infeasible to have every client communicate with every other client
- Auditors (could be other identity providers) can help with these checks
- Client can fetch signed tree roots (STRs) from at least one auditor, chosen at random, and check that they match the STR from the provider
- With more auditors and multiple checks, hard for an attacker to maintain a split view



Auditing for consistency

- How auditors check for linear history:
  - Fetch new STR from identity provider
  - Check signature on new STR
  - Check if hash of last STR seen by the auditor is included in new STR (Need this check to verify that history has not been “forked”)
  - Store new STR for auditing in next epoch

What happens when user goes offline

- If a client goes offline, it can “catch up” by checking a consistency proof for each epoch it was offline

Limitations of CONIKS

Security properties:

- If provider and many/all auditors are compromised, then attacker can present different views to different clients
- If the client goes offline, the client will not detect bad key bindings as they occur
- Relies on an out-of-band whistleblowing mechanism where users and auditors can provide proof of misbehavior

Performance properties

- Large server state (need to store each version of tree) — solved in later work
- Client has to do work for each epoch offline — more efficient in later work
- Shorter epochs mean faster update completion, but more client auditing

SEEMless

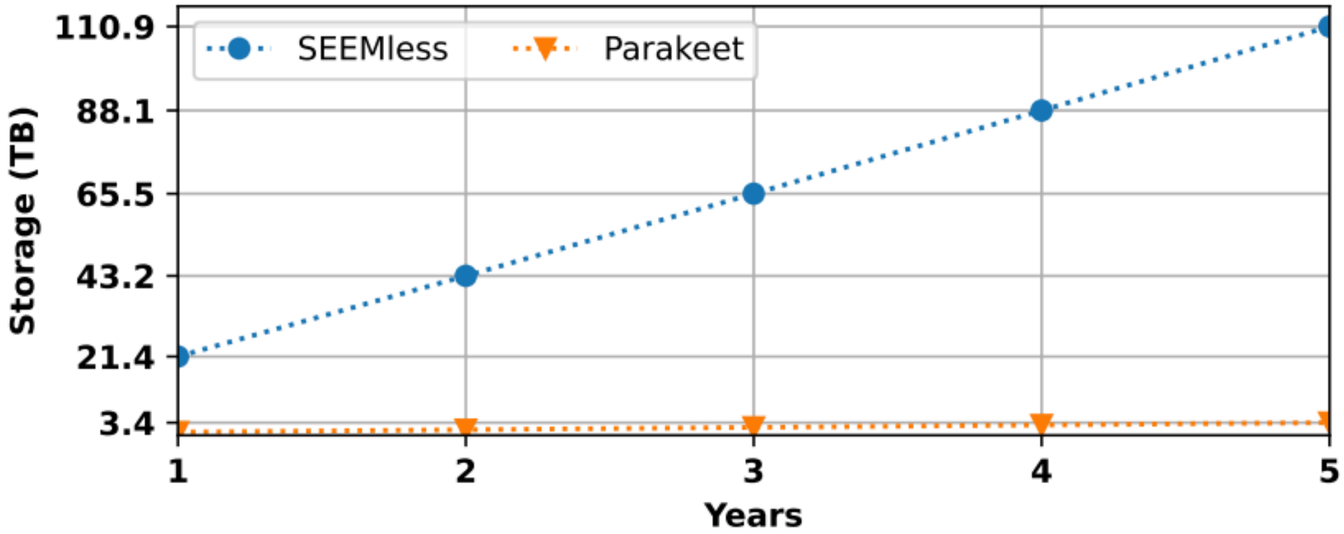
Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, Harjasleen Malvai

- Reduces overhead of client consistency proof checks:
  - CONIKS requires long epochs, which lead to slow updates, because the client must perform consistency checks every epoch.
  - In SEEMless, clients do not need to check a consistency proof for each offline epoch, so can set shorter epoch times (and reduces client work)
- Improves on server storage:
  - CONIKS requires the server to store a tree for every epoch (wasteful because there is redundant information).
  - SEEMless data structure size is proportional to total number of updates instead of epochs

Parakeet

Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Ozturk, Kevin Lewi, Sean Lawlor

- Infinitely growing server data is not practical
- Introduces compaction technique for removing old entries that are no longer useful



Based on estimated WhatsApp costs

References

Chase, M., Deshpande, A., Ghosh, E., & Malvai, H. (2019, November). Seemless: Secure end-to-end encrypted messaging with less trust. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security* (pp. 1639-1656).

Malvai, H., Kokoris-Kogias, L., Sonnino, A., Ghosh, E., Oztürk, E., Lewi, K., & Lawlor, S. (2023). Parakeet: Practical key transparency for end-to-end encrypted messaging. *NDSS 2023.*

Melara, M. S., Blankstein, A., Bonneau, J., Felten, E. W., & Freedman, M. J. (2015). CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)* (pp. 383-398).