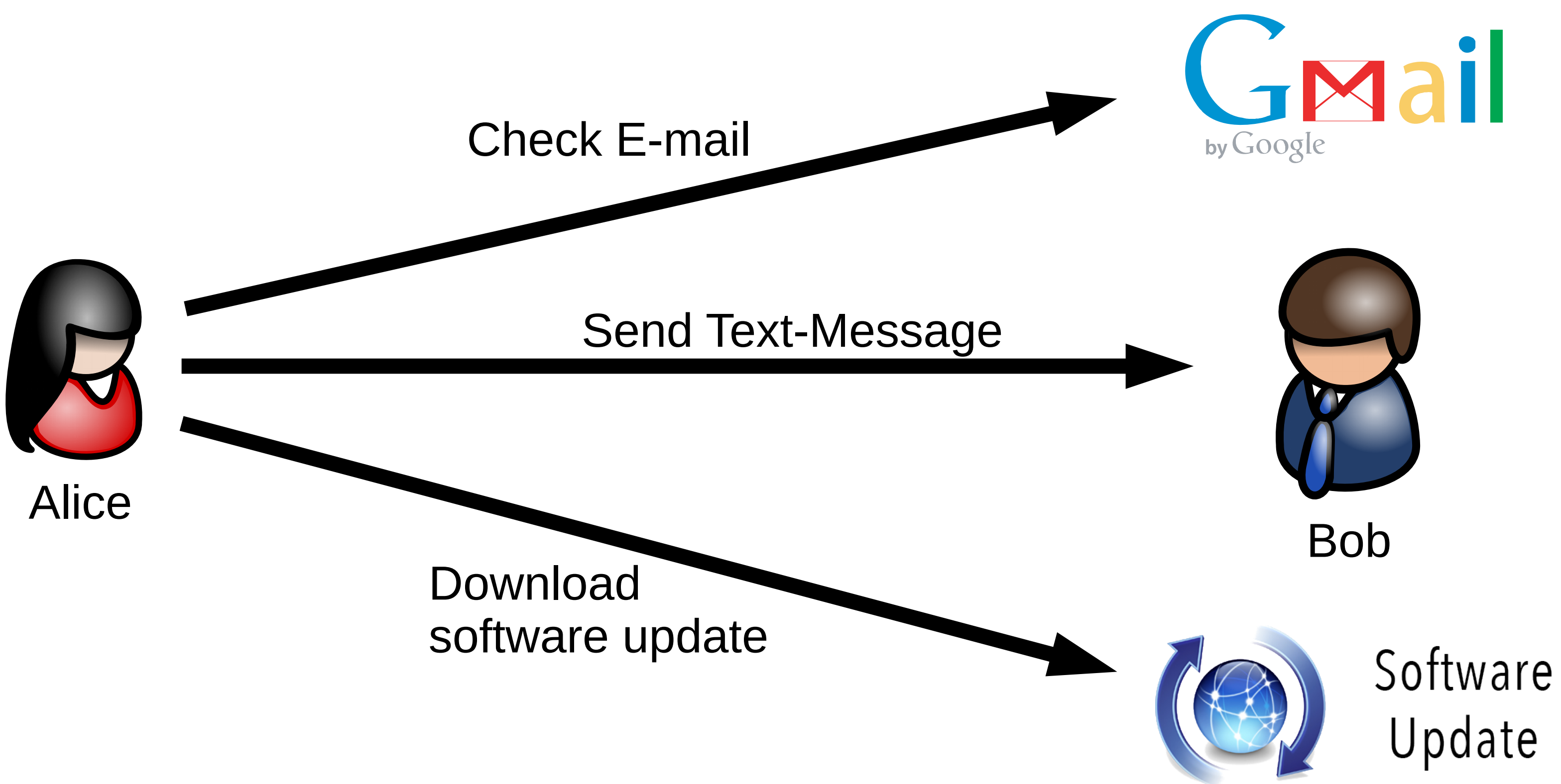
Decentralizing Authorities (such as CT log servers)

http://datatracker.ietf.org/doc/draft-ford-trans-witness/
http://arxiv.org/abs/1503.08768
https://github.com/DeDiS/cothority

Ewa Syta, Iulia Tamas, Dylan Visher, David Wolinsky – **Yale University**
Bryan Ford, Linus Gasser, Nicolas Gailly – **Swiss Federal Institute of Technology (EPFL)**

IETF – November 2, 2015

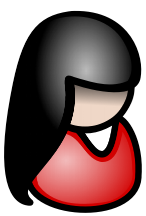
Why do we have authorities?



Why do we have authorities?

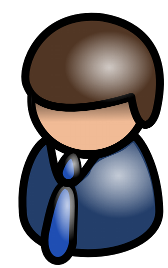
Respect my Authoritah!

Gmail by Google



Alice

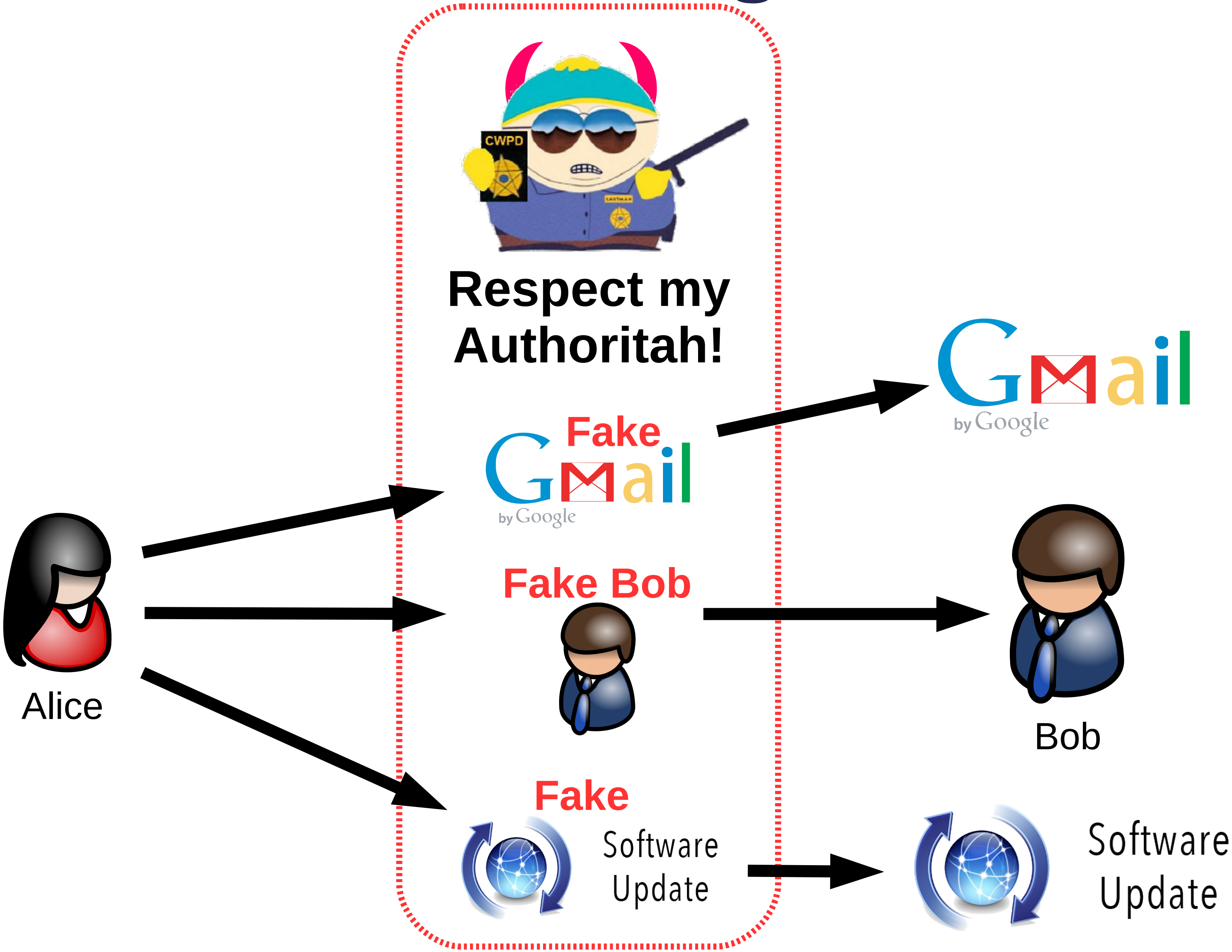
What is:
- Gmail's SSL public key?
- Bob's IM public key?
- Latest version of App?



Bob

Software Update

When authorities go bad

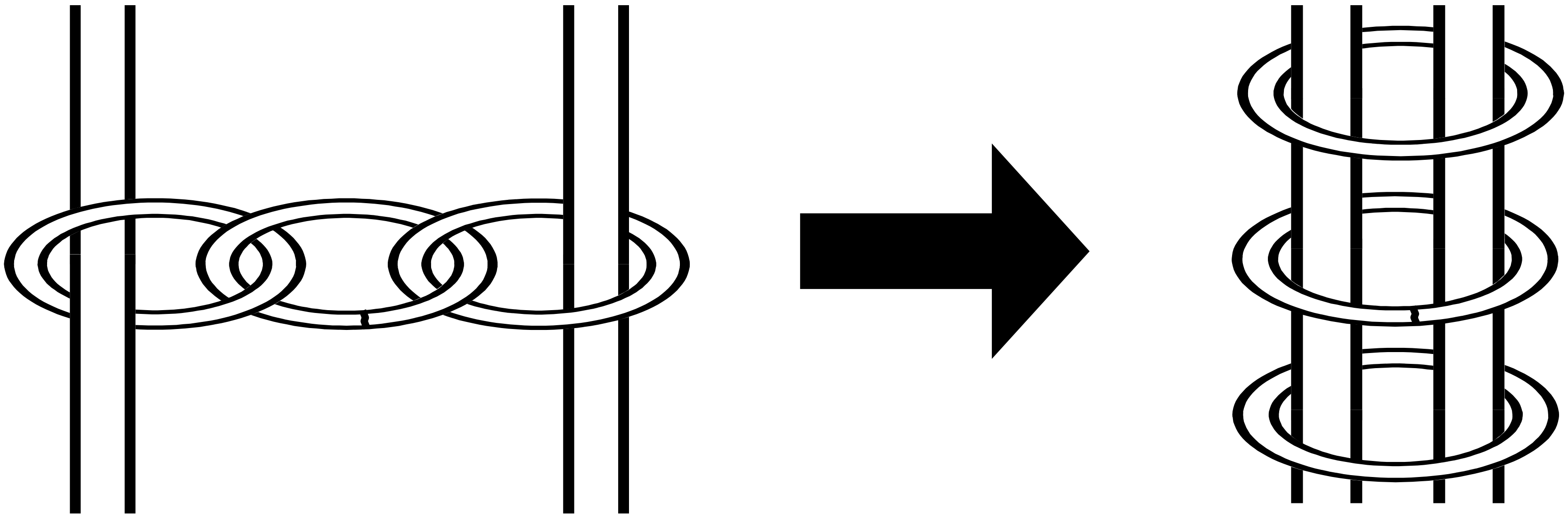


Challenge: Decentralize Authorities

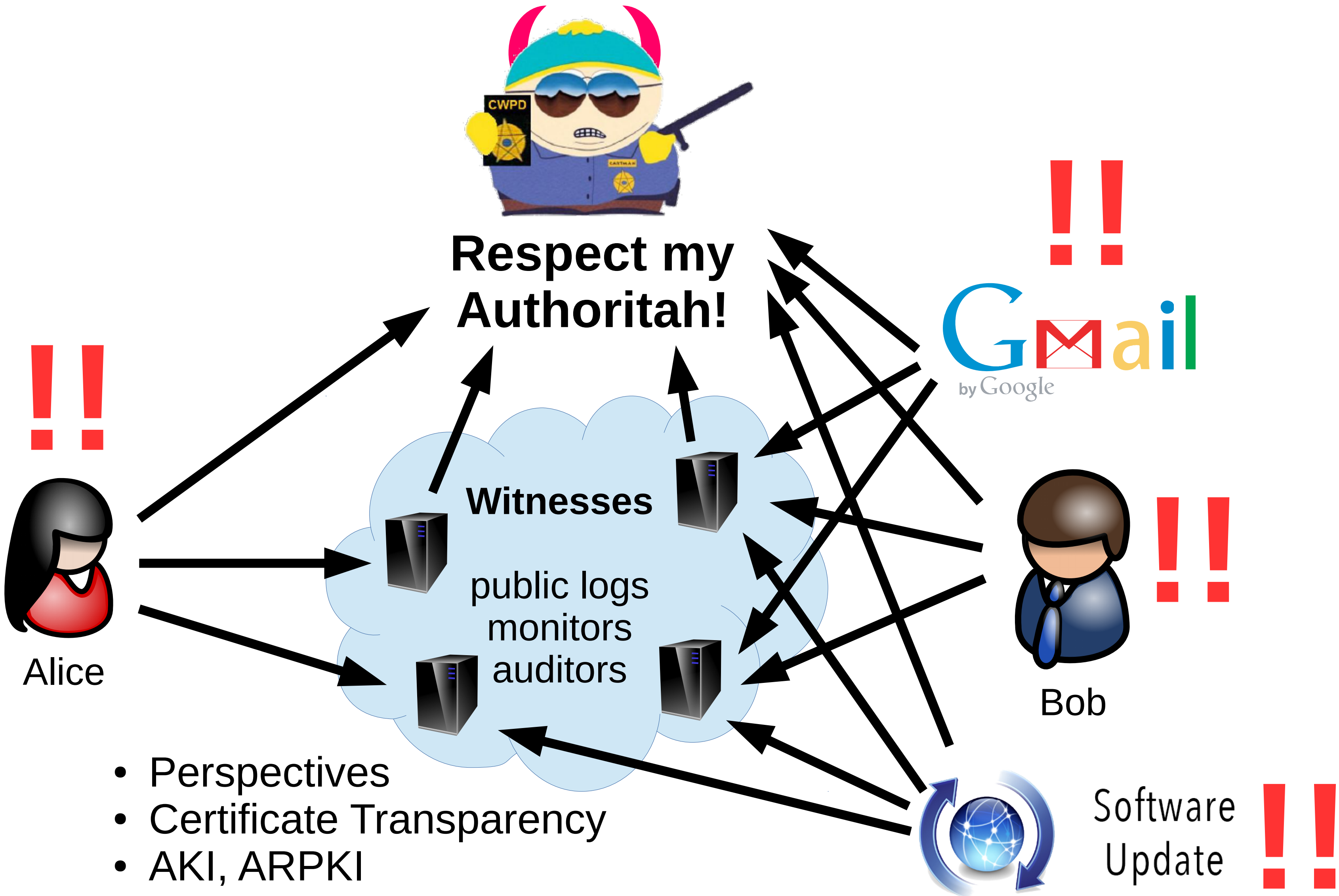
Split important authority functions across multiple participants (preferably independent)

- So authority isn't compromised unless multiple participants compromised

From **weakest-link** to **strongest-link** security

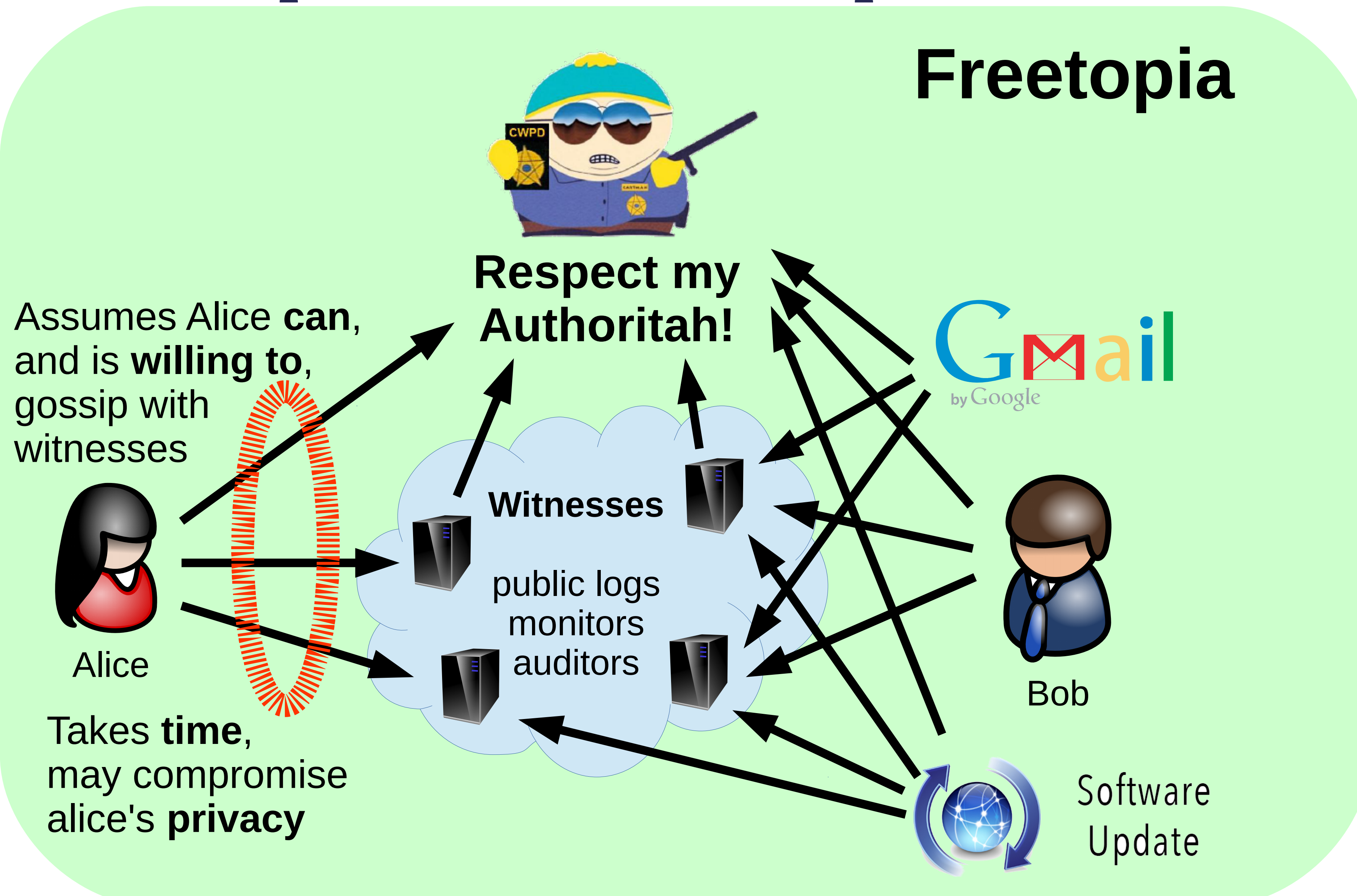


Current Transparency Solutions

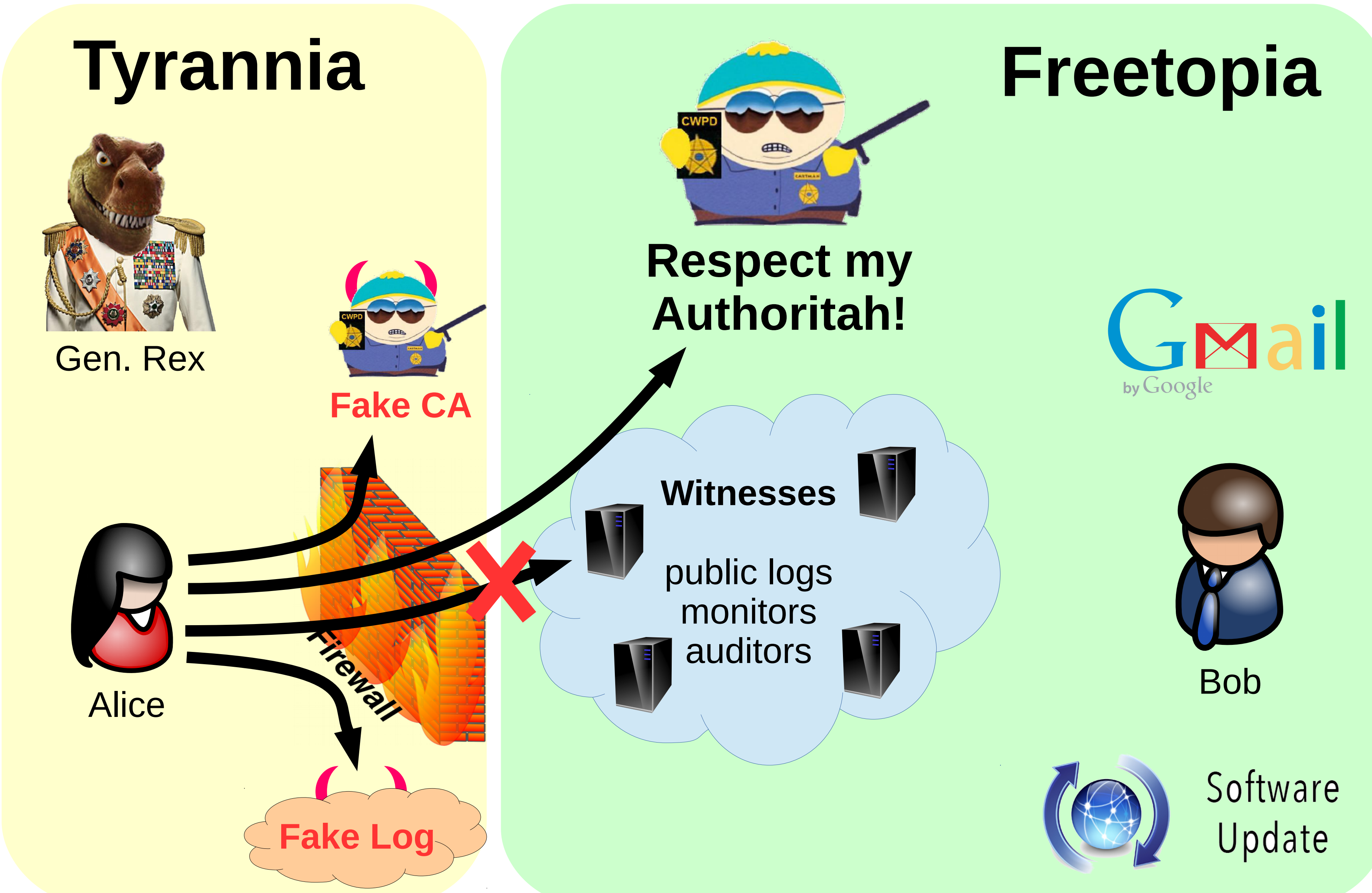


- Perspectives
- Certificate Transparency
- AKI, ARPKI
- CONIKS

An Important Assumption



A Different Scenario



Limitations of Gossip

Detection relies on clients to gossip, but

- Client must be **able** to gossip
  - May fail if attacker controls client's network: compromised WiFi cafe, state-controlled ISP
- Client must be **willing** to gossip
  - Creates privacy issues for clients
- Client must **have time** to gossip
  - Can't delay page load times → attack windows; bigger problem if CT used for software updates!
- Client must **maintain state** to gossip
  - Fails if client is amnesiac, e.g., Tails

Log servers are authorities too

Security is still "weakest-link" across log-servers

Powerful adversary still needs to "acquire" only

- Any one private CA key
- Any two private CT log-server keys ("one Google, one not-Google")

...to silently, secretly MITM-attack victims
by constructing "fake view of CT universe"

3 keys is a better compromise threshold than 1,
but still not as decentralized as we might like!

Towards Proactive Protection

We would like to

- **Proactively** protect clients from attackers using stolen/compromised authority keys
  - Minimize, ideally eliminate vulnerability window
- **Disincentivize** attackers from trying to "acquire" authority's keys in the first place
  - By making them a lot less useful even if acquired

Including CA keys, CT log server keys, DNSSEC keys, NTP time server keys, ...

Protection by Collective Witnessing

"Who watches the watchers?"
Public witnesses!

Clients check authority's signature
and co-signatures of many witnesses

Without communication, client knows:

- Any signed authoritative statement has **already been widely witnessed**
- Any signed authoritative statement conforms to checkable **standards**

Statement could still be bad, but it **won't be secret!**

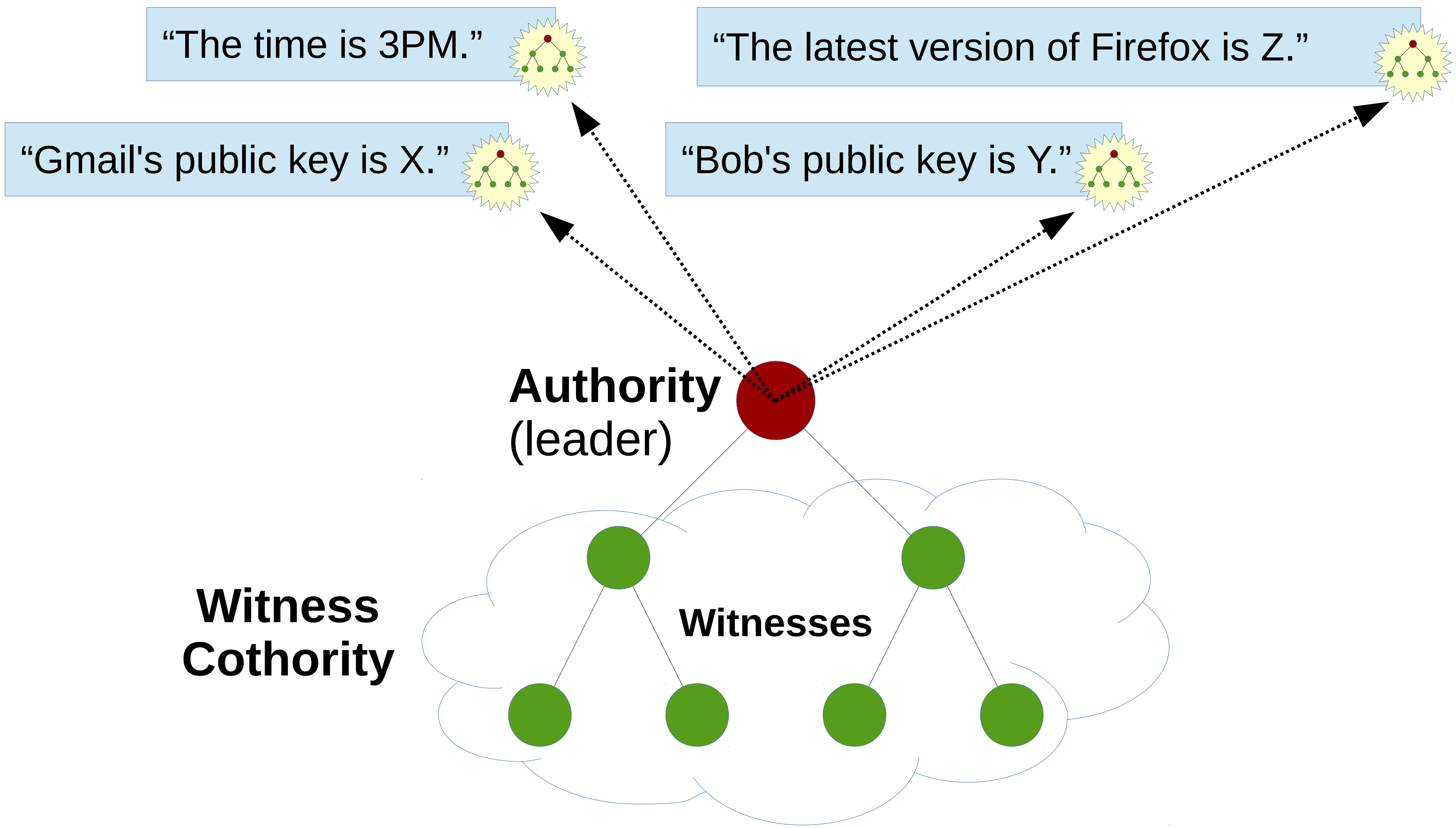
CoSi: Scalable Collective Signing

Semantically like "gathering a list of signatures" but more scalable and efficient:

- **Authority** server generates statements
- **Witness** servers collectively sanity-check and *contribute* to authority's signature
- Each statement gets a **collective signature**: small, quick and easy for clients to verify

→ Authority (or key thief) can't sign anything in secret without *many* colluding followers

CoSi: Scalable Collective Signing



CoSi Crypto Primitives

Builds on well-known primitives:

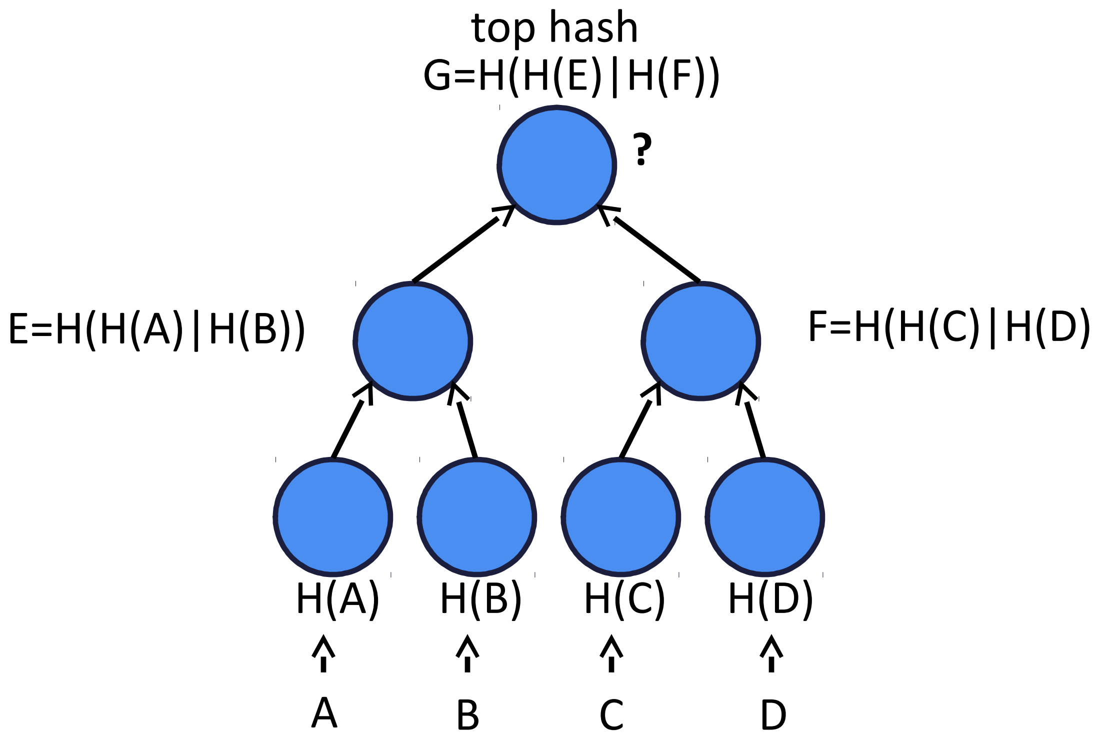
- Merkle Trees
- Schnorr Signature and Multisignatures

CoSi builds upon existing primitives but makes it possible to scale to thousands of nodes

- Using communication trees and aggregation, as in scalable multicast protocols

Merkle Trees

- Every non-leaf node labeled with the hash of the labels of its children.
- Efficient verification of items added into the tree
- Authentication path - top hash and siblings hashes



Schnorr Signature

- Generator *g* of prime order *q* group
- Public/private key pair: $(K=g^k, k)$

	Signer		Verifier
Commitment	$V=g^v$	→	V
Challenge	c	←	$c = H(M\|V)$
Response	$r = (v - kc)$	→	r

Signature on M: (c, r)

Commitment recovery	$V' = g^r K^c = g^{v-kc} g^{kc} = g^v = V$
Challenge recovery	$c' = H(M\|V')$
Decision	$c' = c$? ✓

Collective Signing

- Goal: collective signing with *N* signers
  - Strawman: everyone produces a signature
  - *N* signers-> *N* signatures -> *N* verifications
  - Bad if we have thousands of signers
- Better choice: multisignatures

Schnorr Multisignature

- Key pairs: $(K_1=g^{k_1}, k_1)$ and $(K_2=g^{k_2}, k_2)$

	Signer 1	Signer 2	Verifier	
Commitment	$V_1=g^{v_1}$	$V_2=g^{v_2}$ →	V_1	V_2 $V=V_1*V_2$
Challenge	c	c ←	$c = H(M\|V_1)$	$c = H(M\|V)$
Response	$r_1 = (v_1 – k_1c)$	$r_2 = (v_2 – k_2c)$ →	r_1	r_2 $r=r_1+r_2$

Signature on M: (c, r) (c, r_1) Same signature!

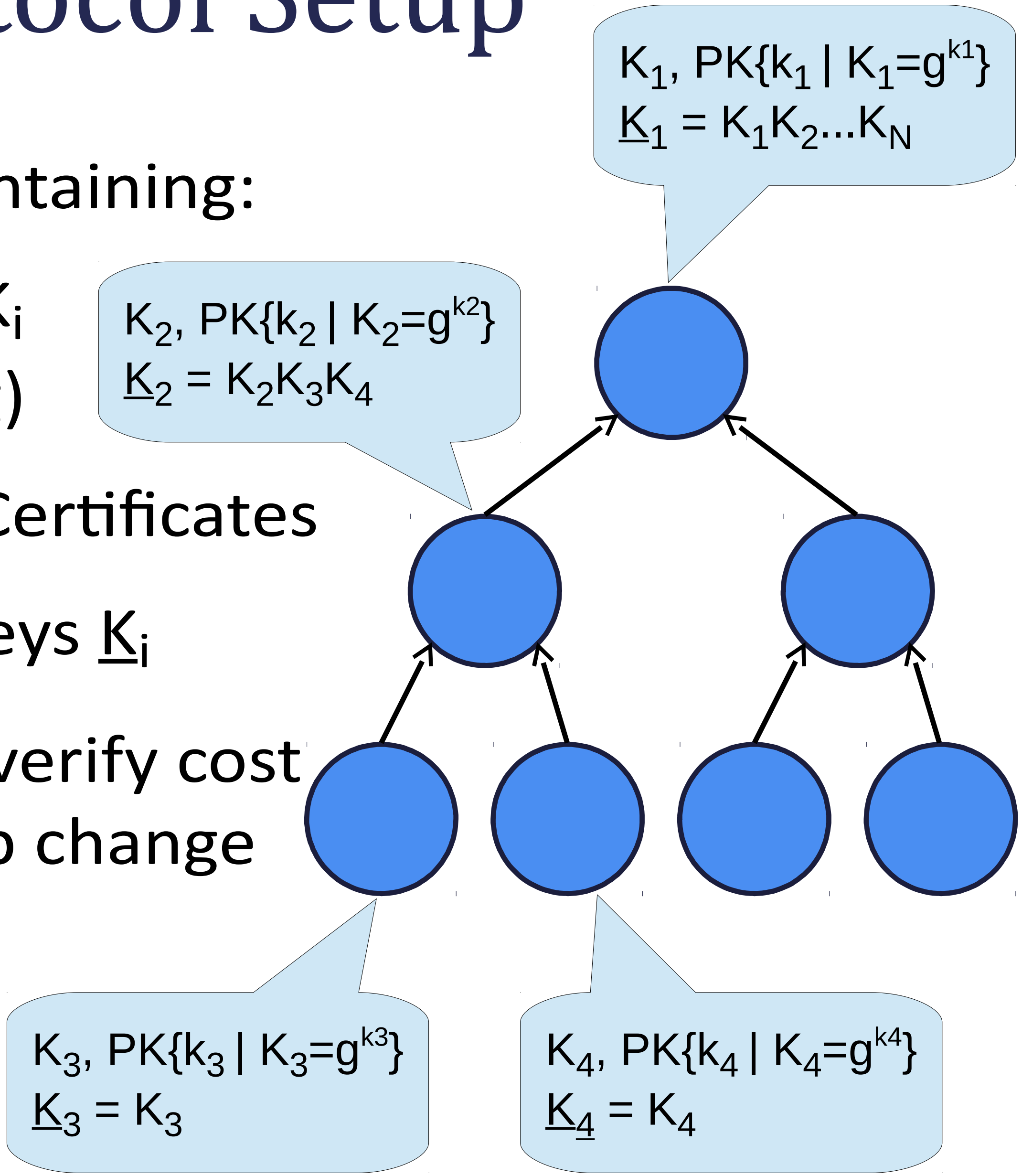
Commitment recovery	Same verification!	$V' = g^rK^c$	$K=K_1*K_2$
Challenge recovery	Done once!	$c' = H(M\|V')$	
Decision		$c' = c$?	

CoSi Protocol Setup

Merkle tree containing:

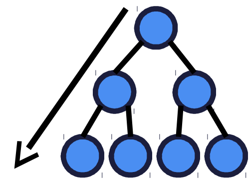
- Public keys K_i (discrete-log)
- Self-signed Certificates
- Aggregate keys $\underline{K}_i$

O(n) one-time verify cost
O(|n'-n|) group change

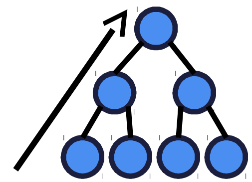


CoSi Protocol Rounds

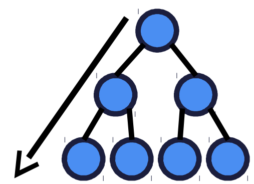
1. Announcement Phase



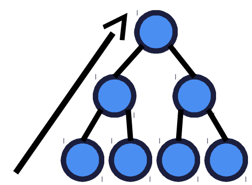
2. Commitment Phase



3. Challenge Phase



4. Response Phase

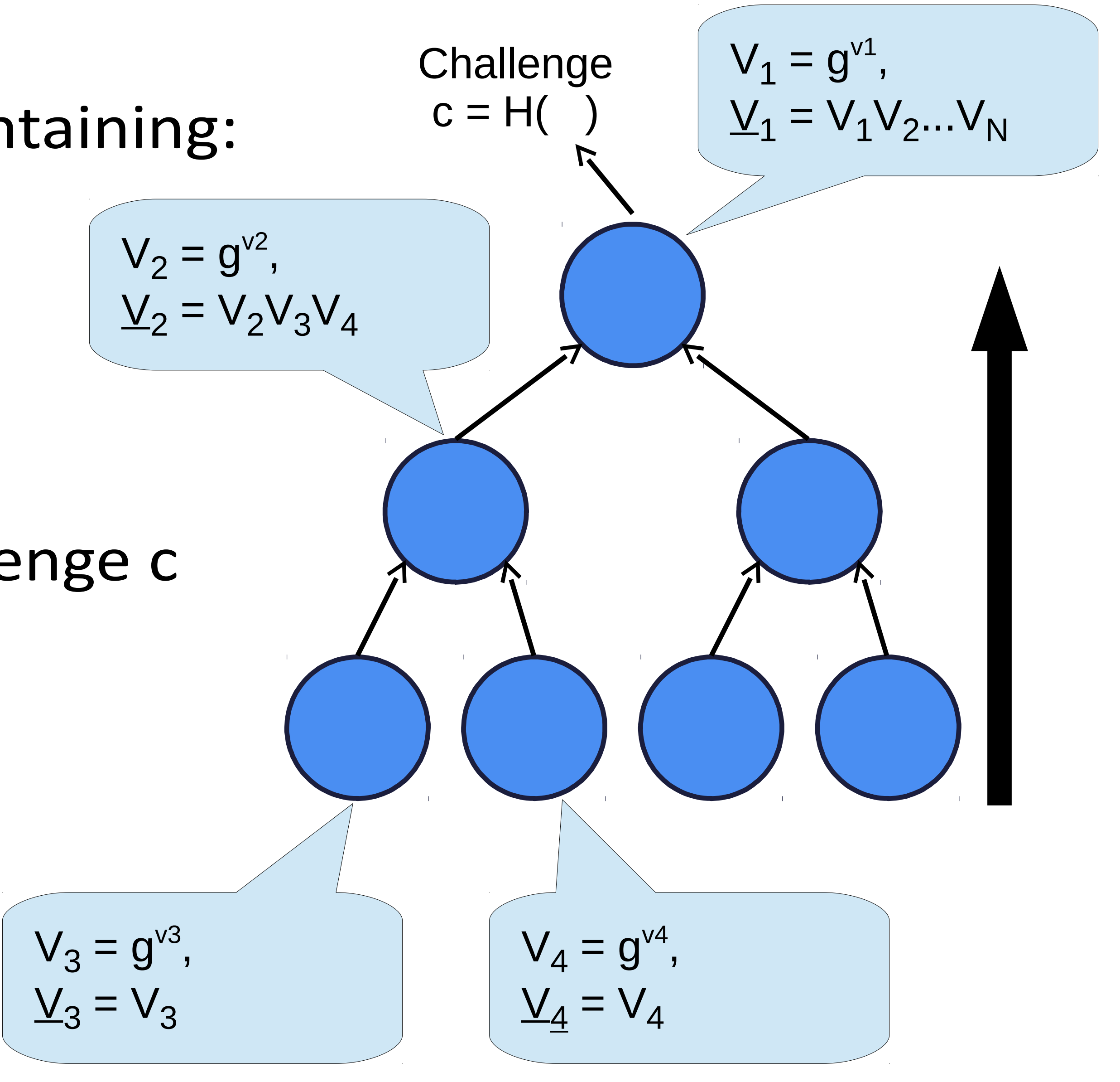


CoSi Commit Phase

Merkle tree containing:

- Commits V_i
- Aggregate commits $\underline{V}_i$

Collective challenge c is **root hash** of per-round Merkle tree



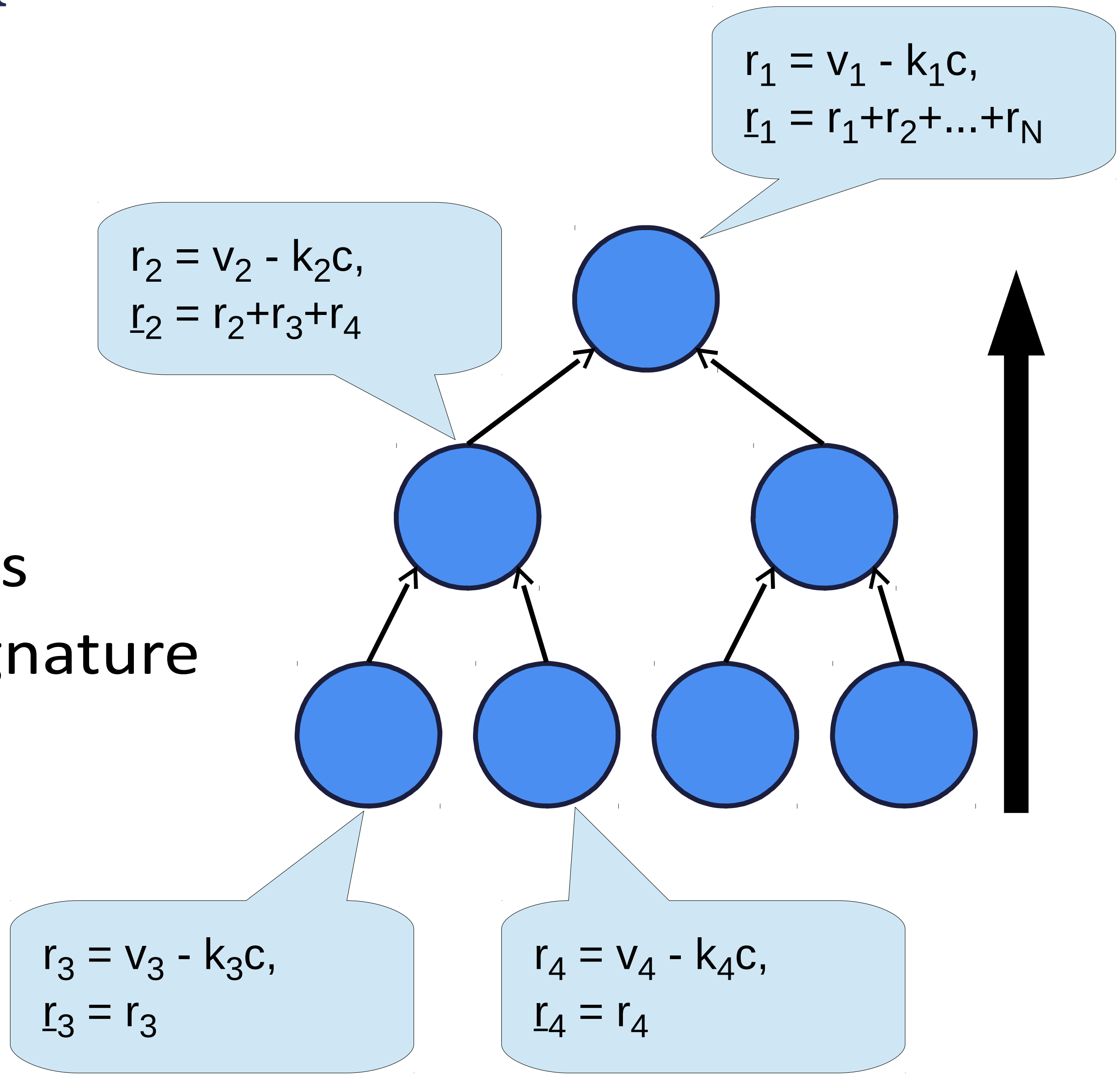
CoSi Response Phase

Compute

- Responses r_i
- Aggregate responses $\underline{r}_i$

Each $(c,\underline{r}_i)$ forms valid **partial** signature

$(c,\underline{r}_1)$ forms **complete** signature



The Availability Challenge

Assume server failures are **rare** but **non-negligible**

- Availability loss, DoS vulnerability if not addressed
- But *persistently bad* servers administratively booted

Two approaches:

- Exceptions – currently implemented, working
- Life Insurance – partially implemented, in-progress

Simple Solution: Exceptions

- If node A fails, remaining nodes create signature
  - For a modified collective key: $K' = K * K^{-1}_A$
  - Using a modified commitment: $V' = V * V^{-1}_A$
  - And modified response: $r' = r - r_A$

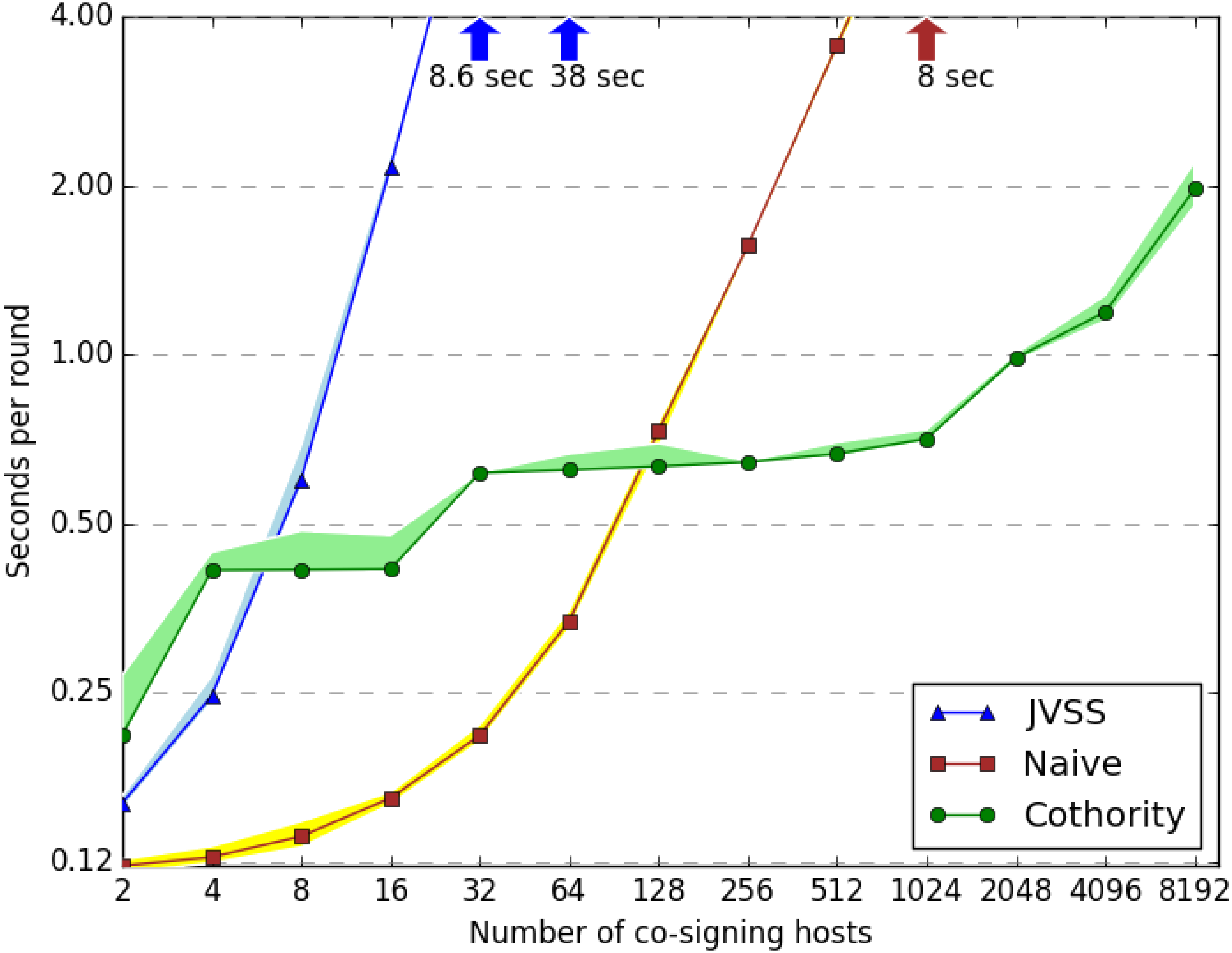
- Client gets a signature under K' along with exception metadata e_A
  - e_A also lists conditions under which it was issued

- Client accepts **only** if a quorum of nodes maintained

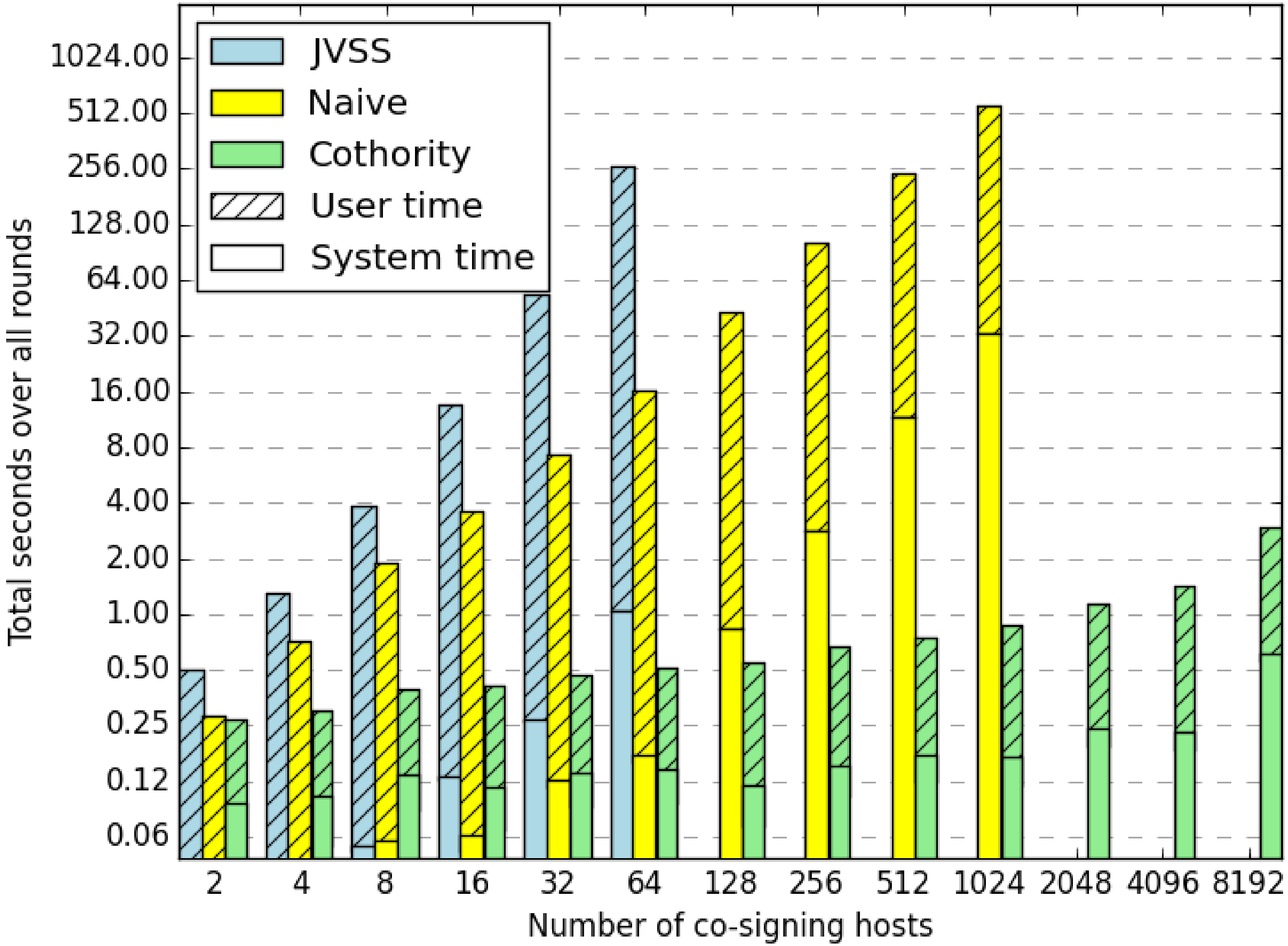
Implementation

- Prototype implementation in Go available
  - https://github.com/DeDiS/cothority
- Performance/scalability testing on DeterLab
  - Up to 8192 virtual CoSi nodes on 64 physical hosts
  - Latency: 100ms roundtrip between two servers
- Preliminary integration into Google CT log server
  - Log server initiates collective signing for STHs, insert collective signature into STH extension field
  - Assumes clients fetch, check STH inclusion proofs (but that's “coming soon” anyway, right?)

Results: Collective Signing Time



Results: Computation Cost



Current Issues and Limitations

CT integration: STH extension semantics

- SthExtensions “covered” by log's signature, but collective signature can't “sign itself”
  - Quick/easy workaround: just collectively sign STH identical except for absence of collective signature
  - But for future, consider class of STH extensions explicitly *not* covered by conventional signature?

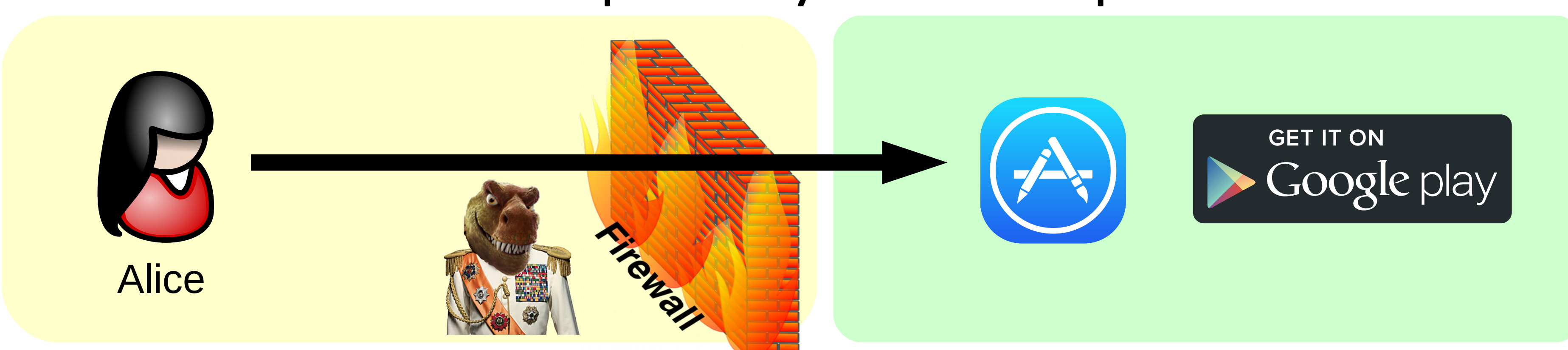
Other current (fixable) limitations

- Tree is more “baked-in” than it should be
- Gaps in both code and documentation

Software Update Scenario

Alice, traveling in Tyrannia, is offered a **software update** for her favorite app

- Claims to be “latest version” - but is it?
- Rex's firewall might inject **authentic** but **outdated, now exploitable** version
- If Alice accepts, she is **instantly Pwned;** retroactive transparency won't help!



Timestamping Cothority

Like classic **digital timestamp** services, only decentralized.

- Each round (e.g., 10 secs):
  1) Each server collects hashes, nonces to timestamp
  2) Each server aggregates hashes into Merkle tree
  3) Servers aggregate local trees into one global tree
  4) Servers collectively sign root of global tree
  5) Server give signed root + inclusion proof to clients
- Clients verify signature + Merkle inclusion proof

Verifiably Fresh Software Updates

Alice accepts only updates with fresh timestamp:

- Knows update can't be an outdated version: tree contains inclusion proof of *her* nonce
- Knows update can't have targeted backdoor: witness cothority ensures *many* parties saw it