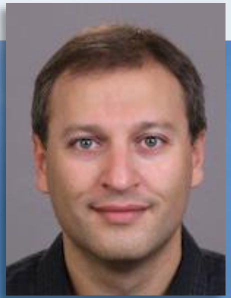
Resolving Concurrency in Group Ratcheting Protocols

Real World Crypto 2021

2020-08-27

Cryptography Group
New York University
Alexander Bienstock, Yevgeniy Dodis,

Horst Görtz Institute for IT Security
Chair for Network and Data Security
Ruhr University Bochum
Paul Rösler



Abstract for RWC Committee

Post-Compromise Security, or PCS, refers to the ability of a given protocol to recover—by means of normal protocol operations—from the exposure of local states of its (otherwise honest) participants. Reaching PCS in group messaging protocols so far either bases on n parallel two-party messaging protocol executions between all pairs of group members in a group of n users (e.g., in the Signal messenger), or on so-called tree based group ratcheting protocols (e.g., developed in the context of the IETF Message Layer Security initiative). Both approaches have great restrictions: Parallel pairwise executions induce for each state update a communication overhead of O(n). While tree-based protocols reduce this overhead to O(log n), they cannot handle concurrent state updates. For resolving such inevitably occurring concurrent updates, these protocols delay reaching PCS up to n communication time slots (potentially more in asynchronous settings such as messaging). Furthermore, a consensus mechanism (such as a central server) is needed in practice.

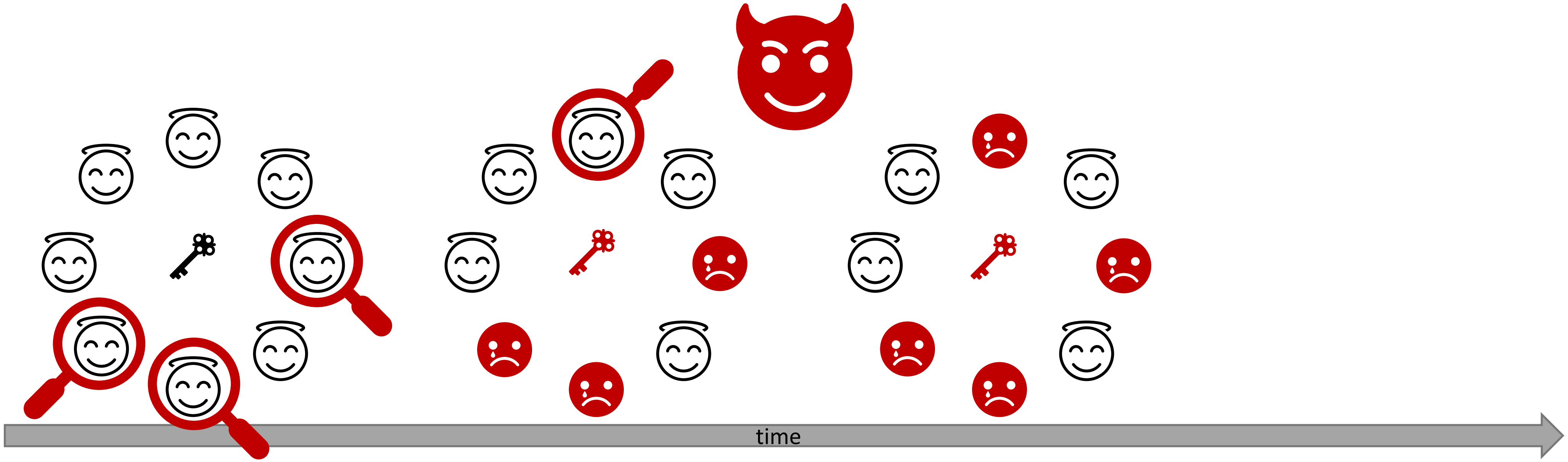
In this talk we discuss the trade-off between PCS, concurrency, and communication overhead in the context of group ratcheting. In particular, we will explain why state updates, concurrently initiated by t group members for reaching PCS immediately, necessarily induce a communication overhead of Ω(t) per message. This result is based on an analysis of generic group ratcheting constructions in a symbolic execution model. Secondly, we will present a new group ratcheting construction that resolves the aforementioned problems with concurrency but reaches a communication overhead of only O(t·(1+log(n/t))), which smoothly increases from O(log n) with no concurrency, to O(n) with unbounded concurrency. Thus, we present a protocol in which each group member can (nearly) immediately recover from exposures independent of concurrency in the group with almost minimal communication overhead. We believe that this result, beyond its applicability to the IETF Message Layer Security (MLS) standardization effort, more generally and more importantly is of interest for (distributed) messaging environments where concurrency is unavoidable.

Although all three considered properties (fast recovery from exposures, little induced communication, and handling of concurrency) are indeed desired by practical messengers, our short review of current real-world protocols and academic proposals at the beginning of this talk reveals (that and) where these approaches fail. Hence, our results, if being deployed, can enhance messaging for a large audience.

While the formal execution of our results is theoretic and partially complex, the high-level ideas and concepts, summarized in this talk, are simple and intuitive. We think that our plain results are interesting for practitioners and the combination of different theoretic approaches to derive these results are insightful to real-world crypto researchers.

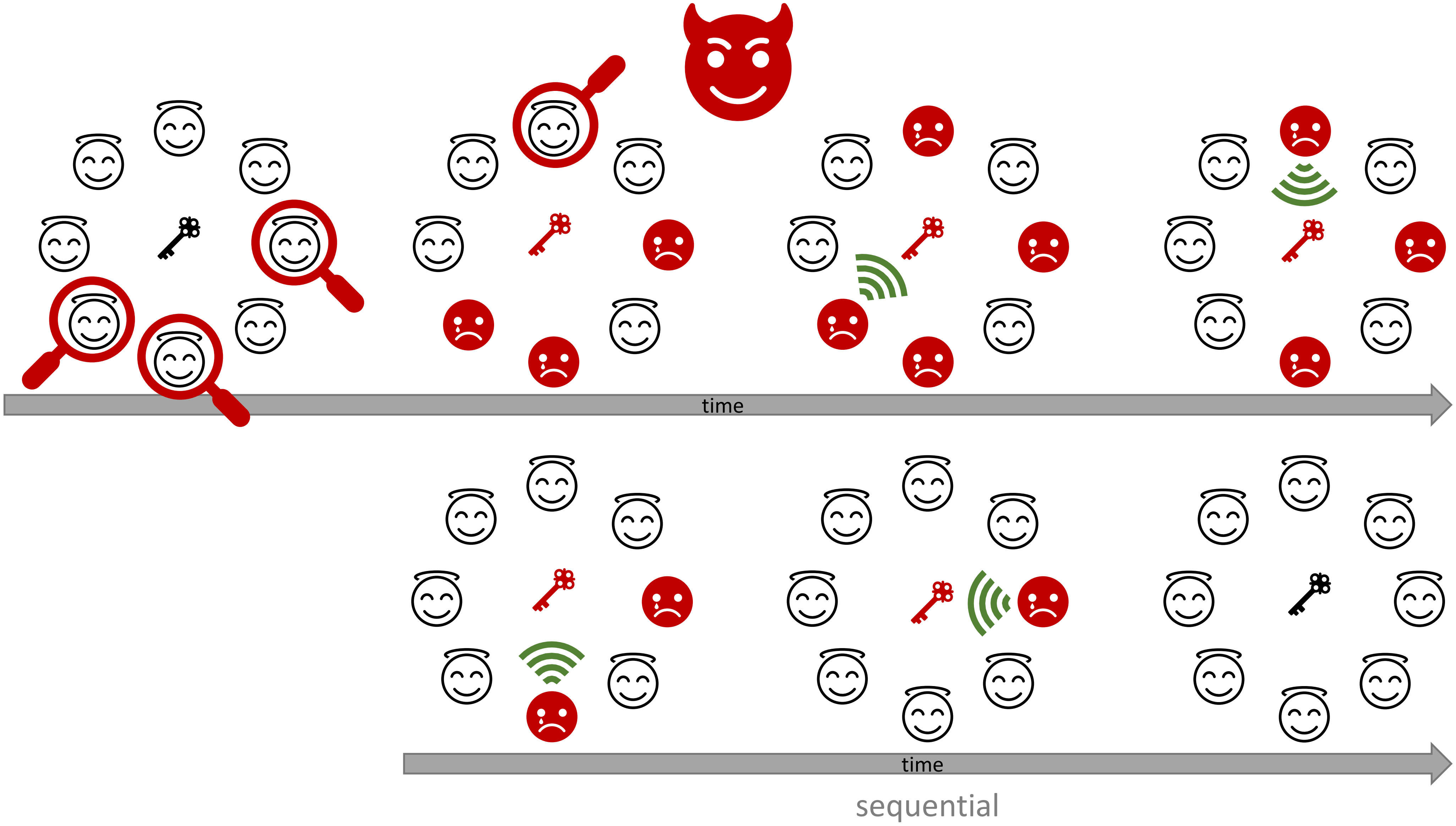
Our primary submission are the presentation slides. For further details and background information, imparted in the talk but maybe not entirely clear from only the slides, we provide a short extended abstract at https://drive.google.com/file/d/1MFlm-P8tZNkK1jxonOZ2yudZ5iA6fmT5/ .

(Concurrent) Group Ratcheting



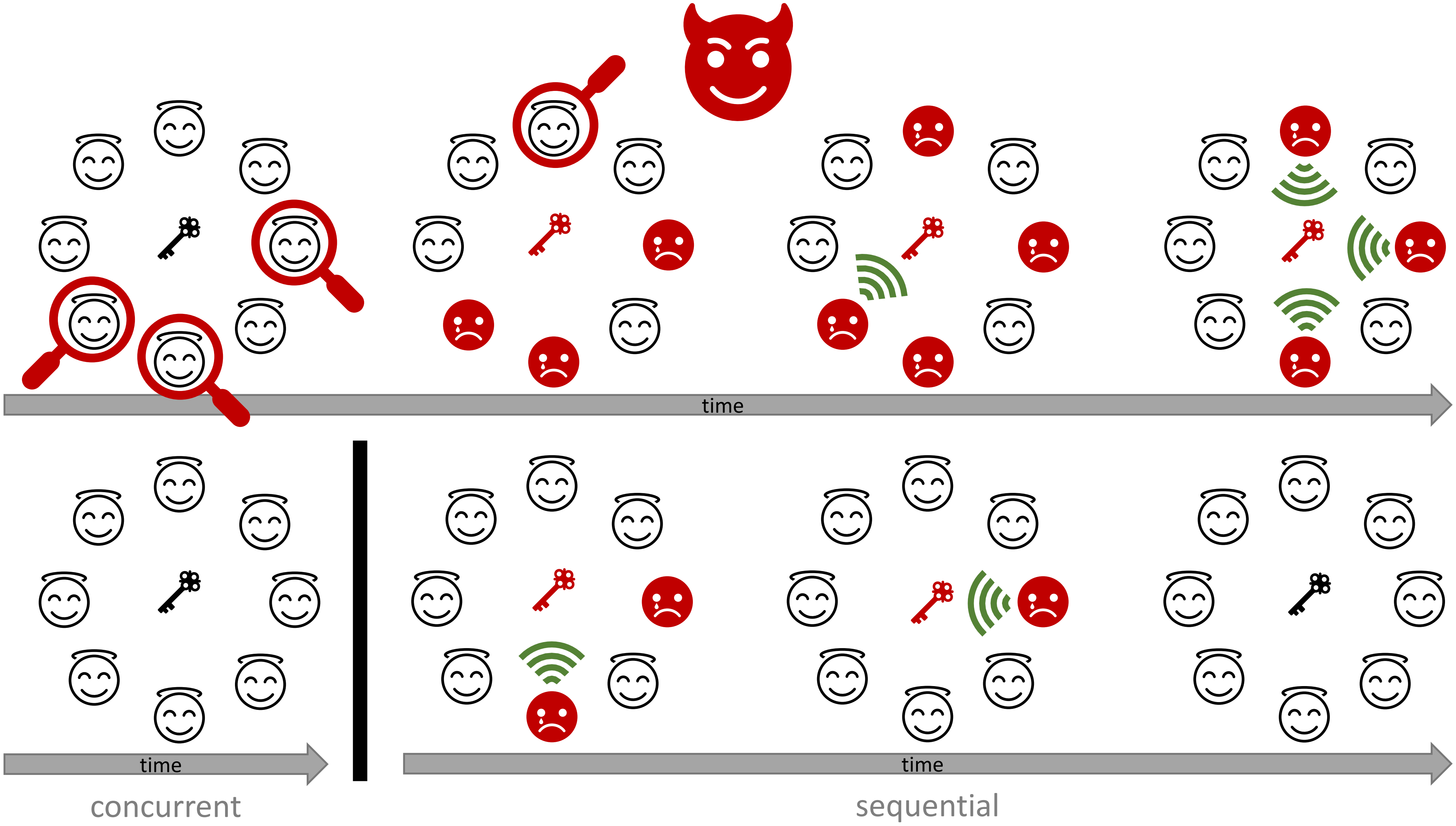
- Group computes joint key**s**
- Exposure of local state temporarily
  - Long-term sessions, mobile devices etc.
  - Leaks group key until all states recovered

(Concurrent) Group Ratcheting



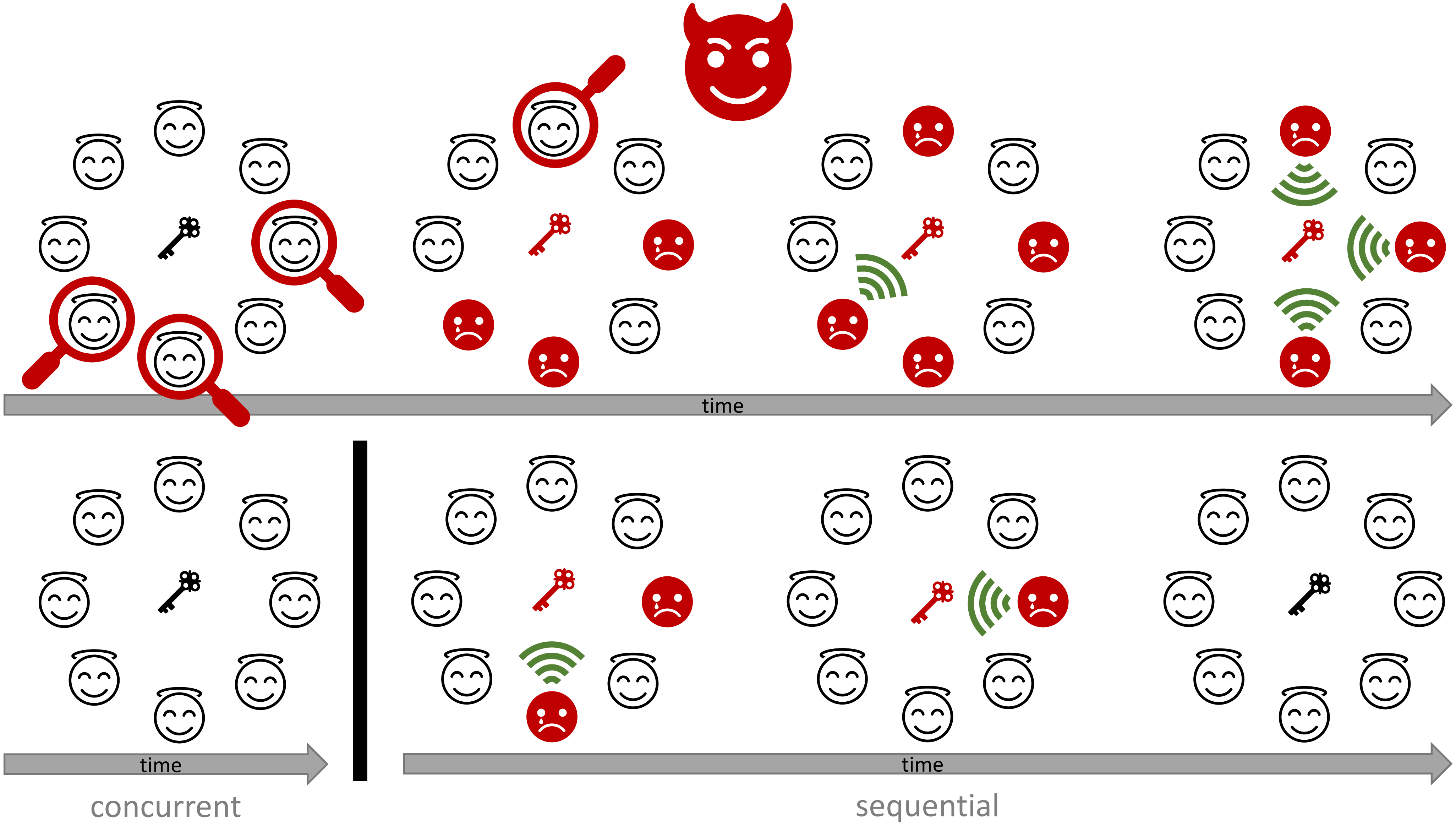
- Group computes joint key**s**
- Exposure of local state temporarily
  - Long-term sessions, mobile devices etc.
  - Leaks group key until all states recovered
- Recovery:
  - Generate new secrets
  - Share public values

(Concurrent) Group Ratcheting



- Group computes joint keys
- Exposure of local state temporarily
  - Long-term sessions, mobile devices etc.
  - Leaks group key until all states recovered
- Recovery:
  - Generate new secrets
  - Share public values
- Concurrent recovery
  - Speedup
  - Merge recoveries

(Concurrent) Group Ratcheting



Target:

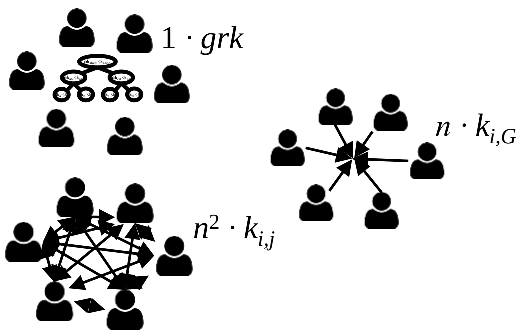
1. Post-Compromise Security
2. Small shares
3. Concurrency

Otherwise:

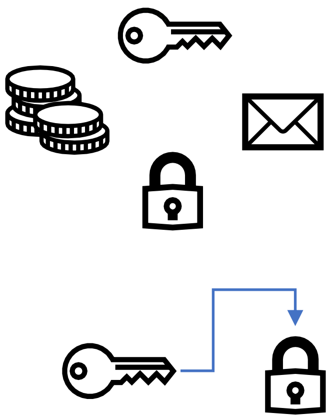
- Slow recovery from exposures
- Consensus required

→ Inapplicable to decentralized networks

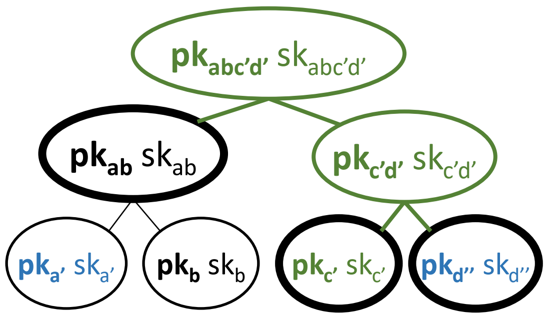
Agenda



Previous Work: What's the Problem? → Lower Bound: What's the minimal overhead?



Upper Bound: Almost optimal construction → Open Questions …

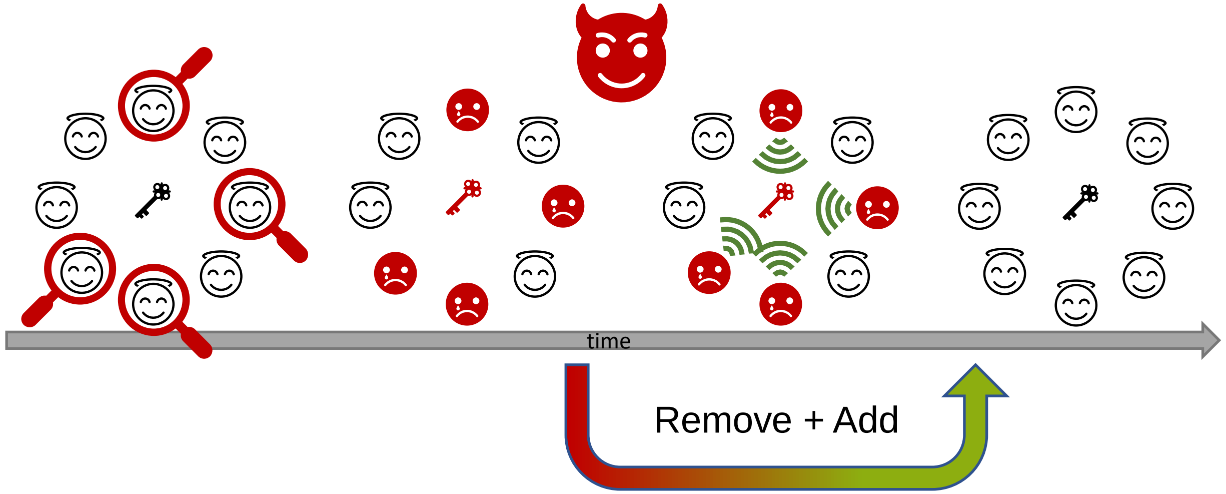


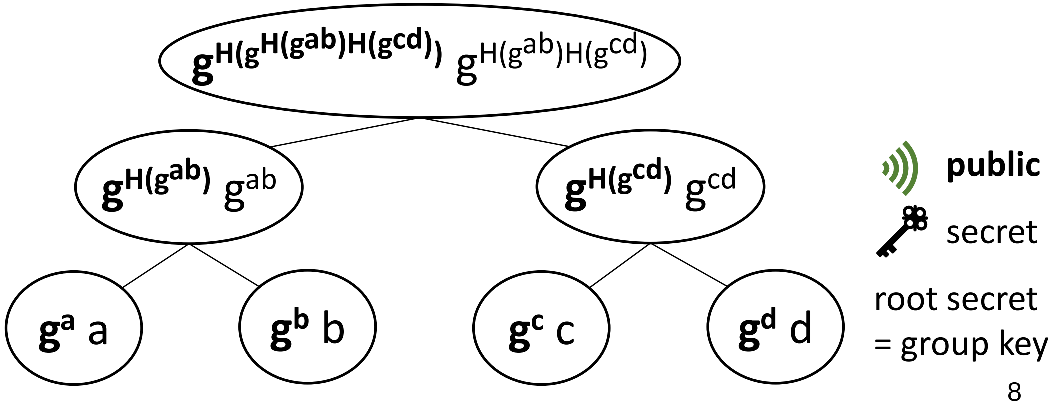
Ω(t) vs. O(t·(1+log(n/t))?
NIKE?
PCS-Delay?
Forward-secrecy?
Application to MLS

Previous Work: What's the Problem?

- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recovery = Remove + Add (R&A)

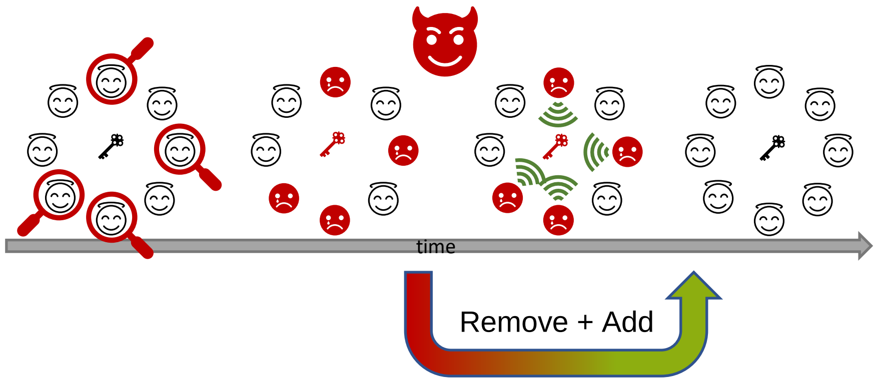
  - Many protocols from '80s – '00ers
  - Tree-based DGKE best suited for asynchronous settings:
    - Little communication for R&A: O(log n)
    - (Almost) non-interactive for R&A
    - → First known DH-tree-based protocol [KPT'04]

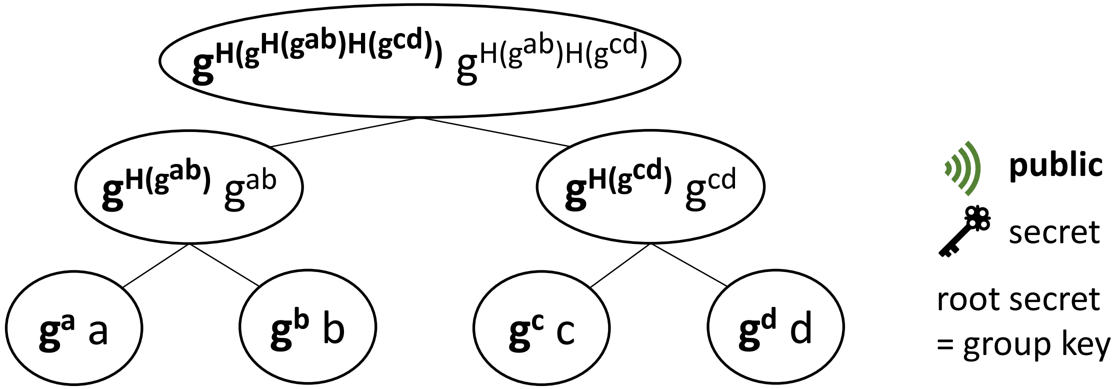




Previous Work: What's the Problem?

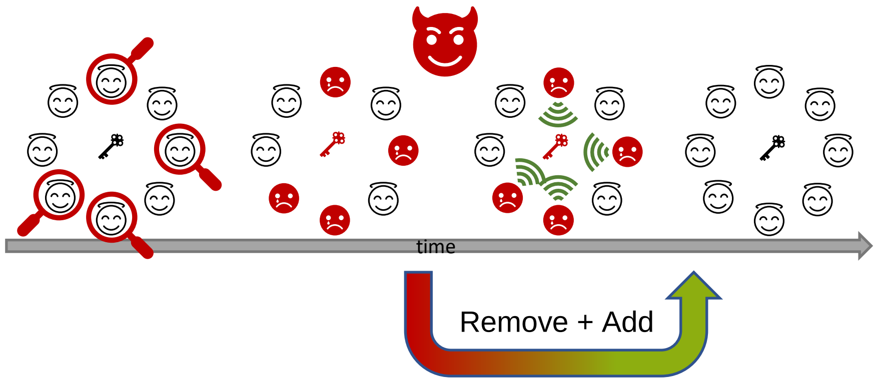
- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recover = Remove + Add (R&A)
  - Tree-based DGKE best suited for asynchronous settings
- Ratcheting in trees
  - Merge R&A [CCGMM'18]

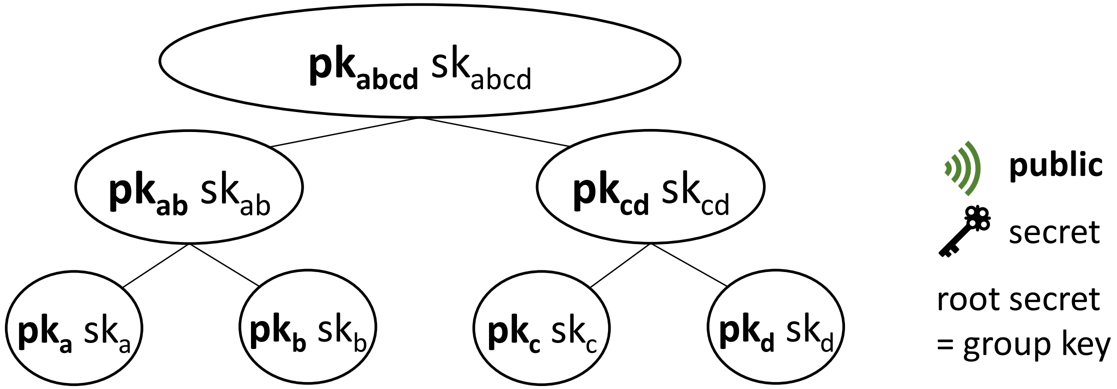




Previous Work: What's the Problem?

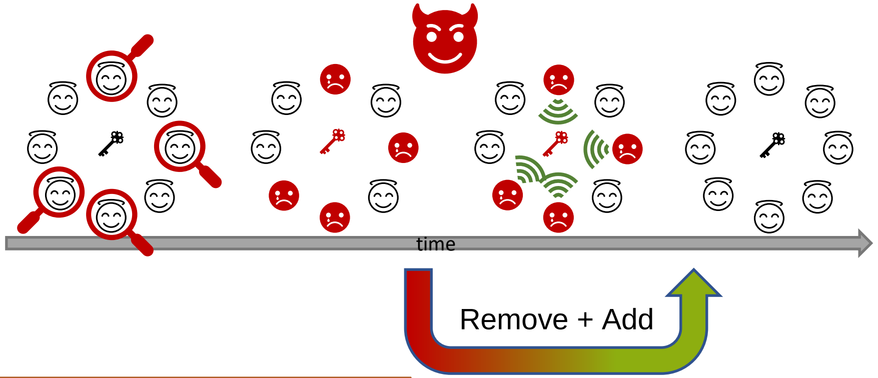
- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recover = Remove + Add (R&A)
  - Tree-based DGKE best suited for asynchronous settings
- Ratcheting in trees
  - Merge R&A [CCGMM'18]
  - DH to KEM [BBR'18]



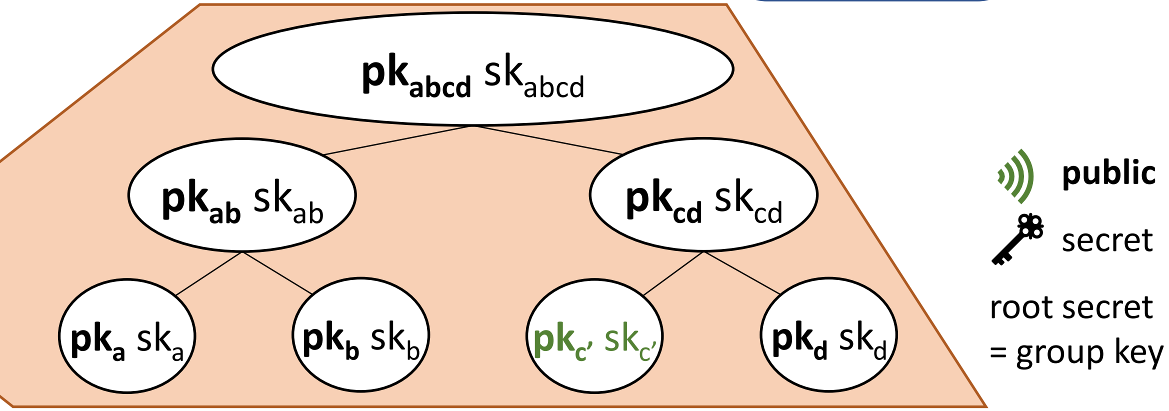


Previous Work: What's the Problem?

- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recover = Remove + Add (R&A)
  - Tree-based DGKE best suited for asynchronous settings

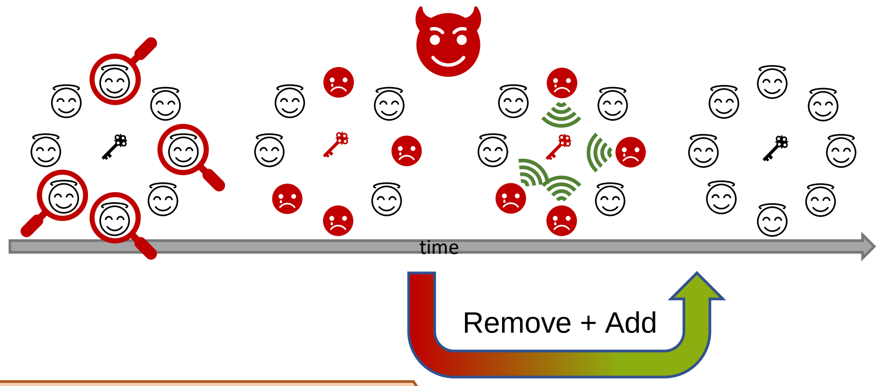


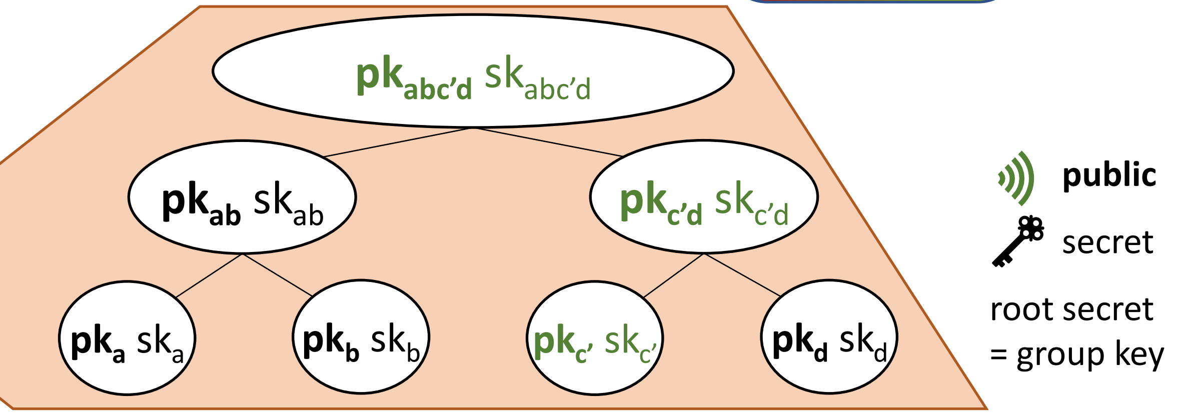
- Ratcheting in trees
  - Merge R&A [CCGMM'18]
  - DH to KEM [BBR'18]
    - Recovery: sample $x_{c'}$, $sk_{c'}=x_{c'}$, $pk_{c'}=gen(sk_{c'})$,



Previous Work: What's the Problem?

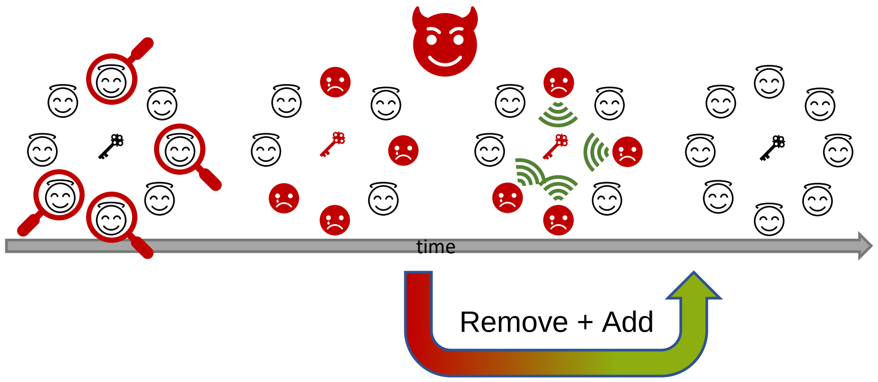
- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recover = Remove + Add (R&A)
  - Tree-based DGKE best suited for asynchronous settings
- Ratcheting in trees
  - Merge R&A [CCGMM'18]
  - DH to KEM [BBR'18]
    - Recovery: sample $x_{c'}$, $sk_{c'}=x_{c'}$, $pk_{c'}=gen(sk_{c'})$, $x_{c'd}=H(x_{c'})$, $enc(pk_d, x_{c'd})$, $sk_{c'd}=x_{c'd}$, …

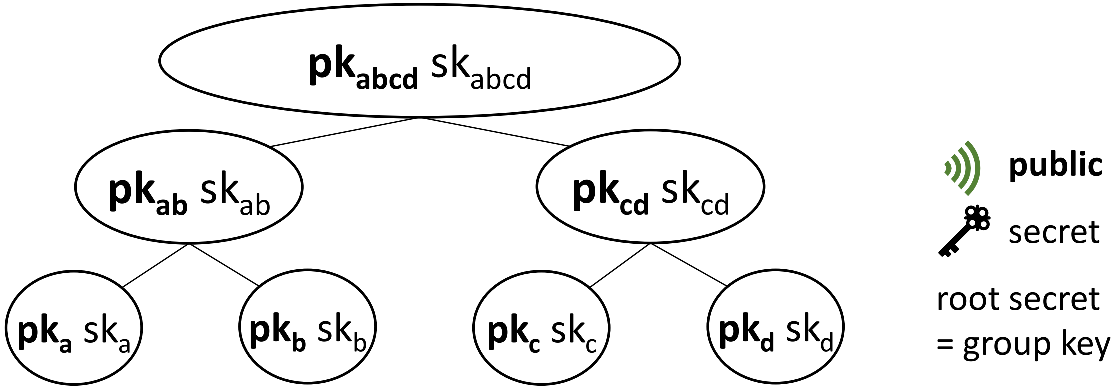




Previous Work: What's the Problem?

- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recover = Remove + Add (R&A)
  - Tree-based DGKE best suited for asynchronous settings
- Ratcheting in trees
  - Merge R&A [CCGMM'18]
  - DH to KEM [BBR'18]
  - Better forward-secrecy [ACDT'20]
  - Maintain balanced tree [ACCKKPPW'19]

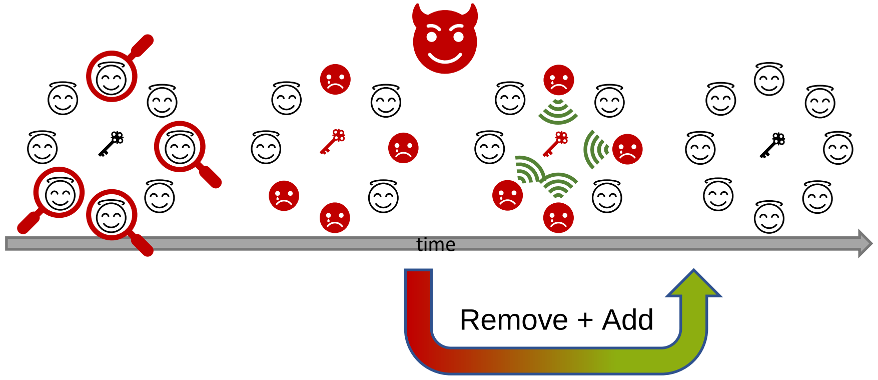


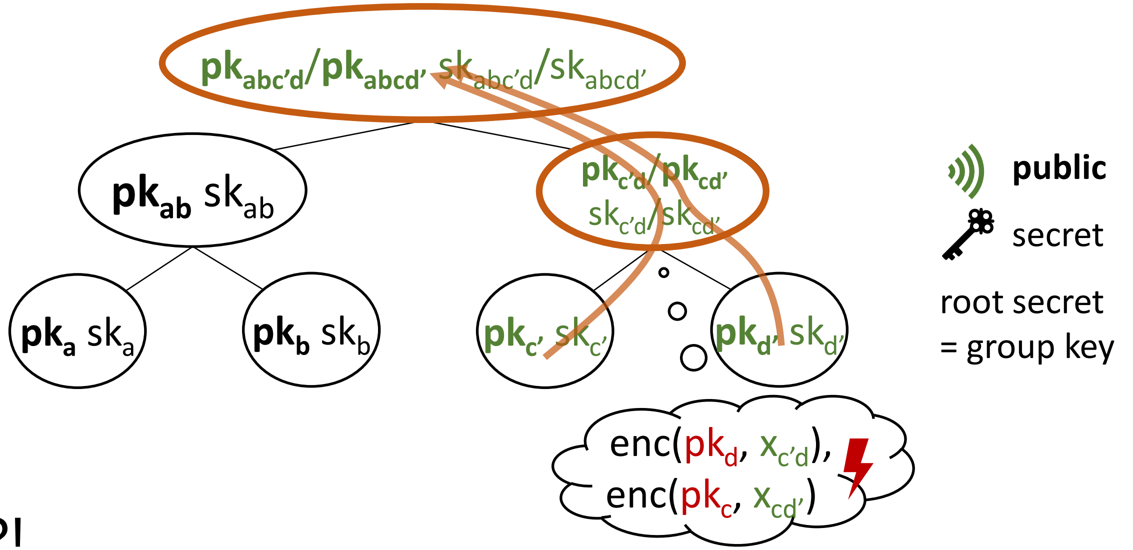


Previous Work: What's the Problem?

- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recover = Remove + Add (R&A)
  - Tree-based DGKE best suited for asynchronous settings
- Ratcheting in trees
  - Merge R&A [CCGMM'18]
  - DH to KEM [BBR'18]
  - Better forward-secrecy [ACDT'20]
  - Maintain balanced tree [ACCKKPPW'19]
- No concurrency
  - Intersection of concurrently updated paths

  → Merging under PCS without multiparty-NIKE?!

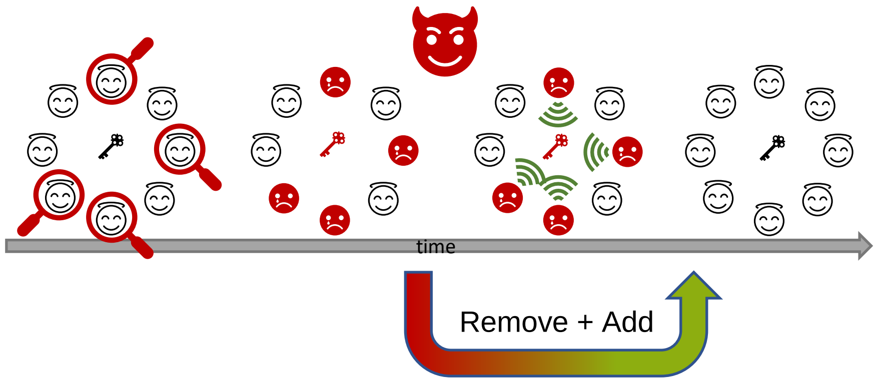


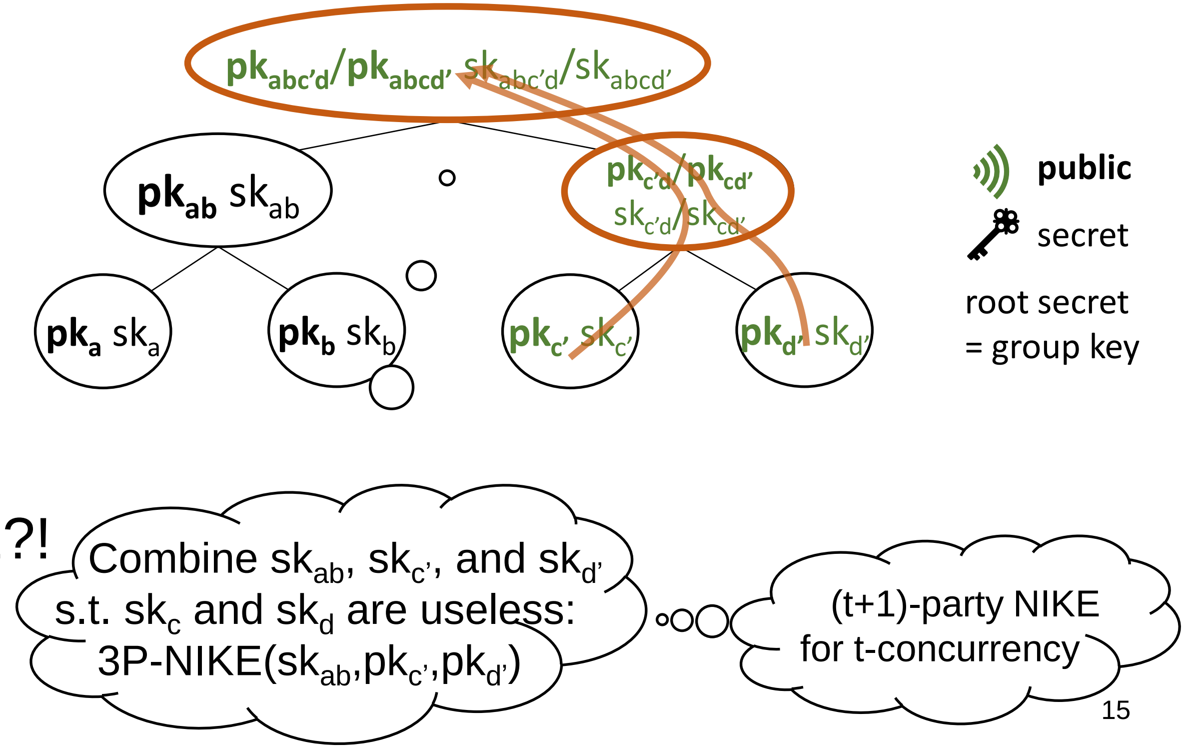


Previous Work: What's the Problem?

- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recover = Remove + Add (R&A)
  - Tree-based DGKE best suited for asynchronous settings
- Ratcheting in trees
  - Merge R&A [CCGMM'18]
  - DH to KEM [BBR'18]
  - Better forward-secrecy [ACDT'20]
  - Maintain balanced tree [ACCKKPPW'19]
- No concurrency
  - Intersection of concurrently updated paths

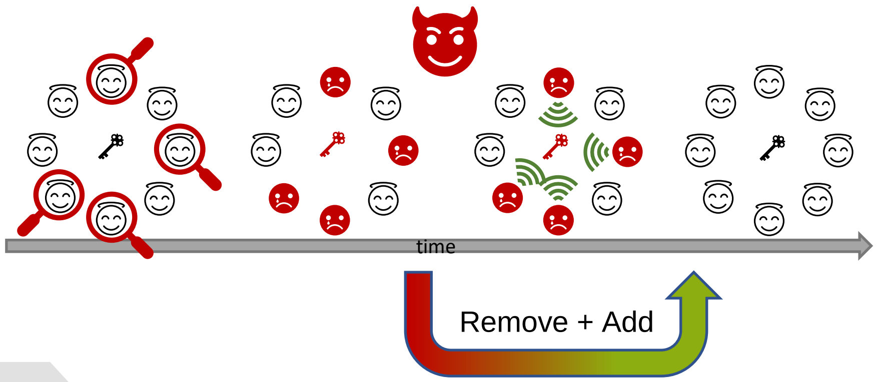
  → Merging under PCS without multiparty-NIKE?!



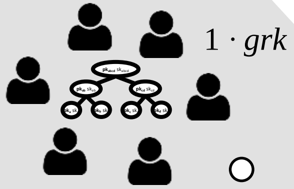


Previous Work: What’s the Problem?

- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recover = Remove + Add (R&A)
  - Tree-based DGKE best suited for asynchronous settings



- Ratcheting in trees
  - Merge R&A [CCGMM’18]
  - DH to KEM [BBR’18]
  - Better forward-secrecy [ACDT’20]
  - Maintain balanced tree [ACCKKPPW’19]



PCS	Overhead	Concurrency
✔	O(log n)	✘

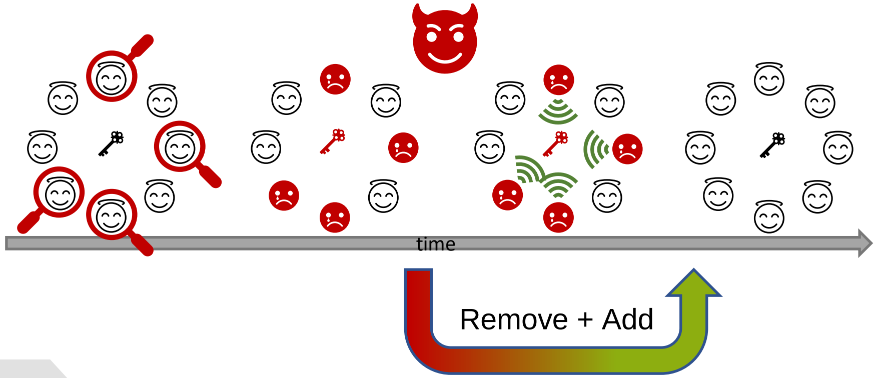
MLSv9 worst-case:

PCS	Overhead	Concurrency
(✔)	O(n)	(✔)

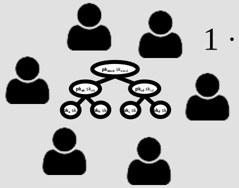
→ Rejects concurrent path updates
→ Degrades to “n-tree”

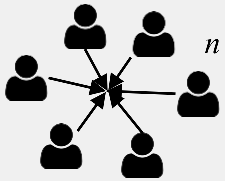
Previous Work: What's the Problem?

- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recover = Remove + Add (R&A)
  - Tree-based DGKE best suited for asynchronous settings



- Ratcheting in trees
  - Merge R&A [CCGMM'18]
  - DH to KEM [BBR'18]
  - Better forward-secrecy [ACDT'20]
  - Maintain balanced tree [ACCKKPPW'19]
- Real-World
  - Forward-secure hash chain [WhatsApp]

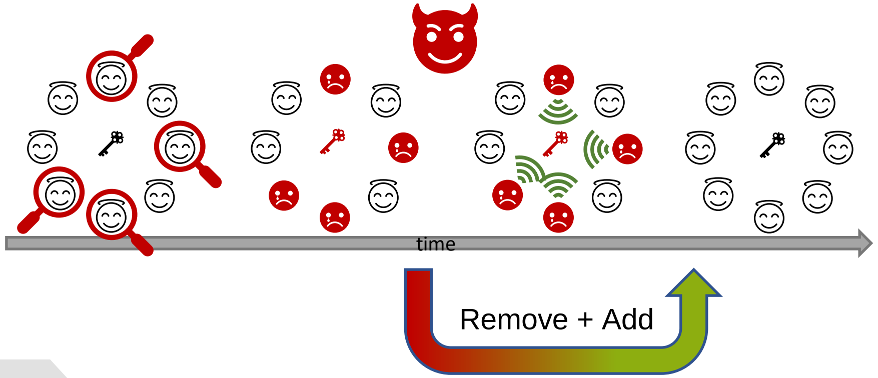




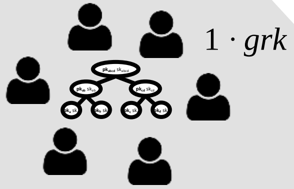
PCS	Overhead	Concurrency
✔	O(log n)	✘
✘	O(1)	✔

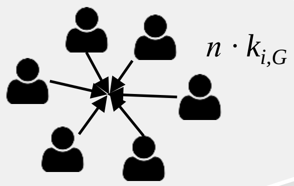
Previous Work: What’s the Problem?

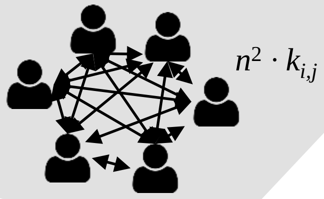
- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recover = Remove + Add (R&A)
  - Tree-based DGKE best suited for asynchronous settings



- Ratcheting in trees
  - Merge R&A [CCGMM’18]
  - DH to KEM [BBR’18]
  - Better forward-secrecy [ACDT’20]
  - Maintain balanced tree [ACCKKPPW’19]
- Real-World
  - Forward-secure hash chain [WhatsApp]
  - Parallel pair-wise communication [Signal]



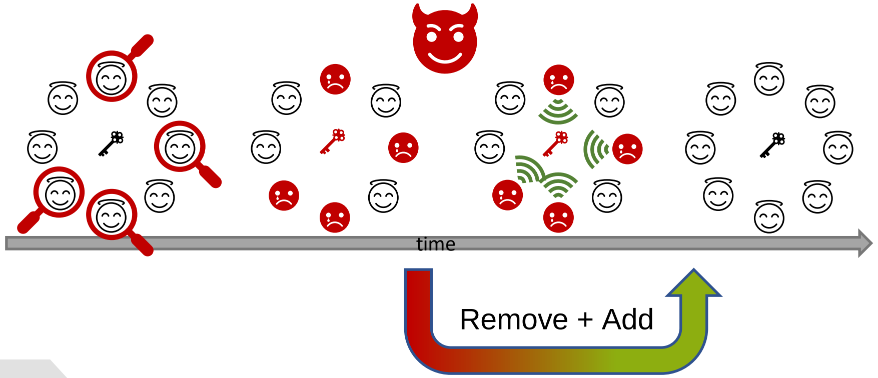




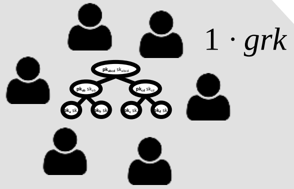
PCS	Overhead	Concurrency
✔	O(log n)	✘
✘	O(1)	✔
✔	O(n)	✔

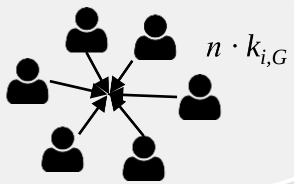
Previous Work: What's the Problem?

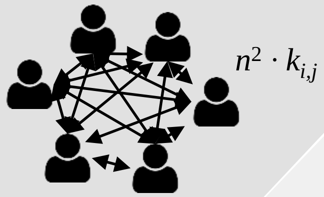
- Essentially: Dynamic group key exchange (DGKE)
  - Expose = Unwanted member
  - Recover = Remove + Add (R&A)
  - Tree-based DGKE best suited for asynchronous settings



- Ratcheting in trees
  - Merge R&A [CCGMM'18]
  - DH to KEM [BBR'18]
  - Better forward-secrecy [ACDT'20]
  - Maintain balanced tree [ACCKKPPW'19]
- Real-World
  - Forward-secure hash chain [WhatsApp]
  - Parallel pair-wise communication [Signal]
- We

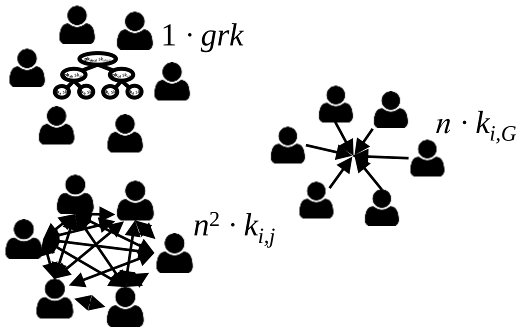






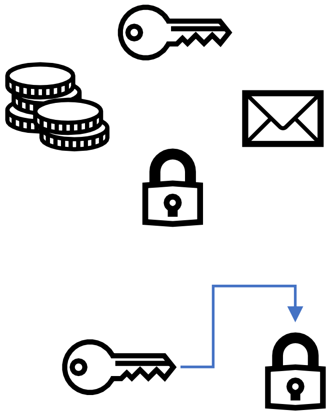
PCS	Overhead	Concurrency
✔	O(log n)	✘
✘	O(1)	✔
✔	O(n)	✔
✔	?	✔

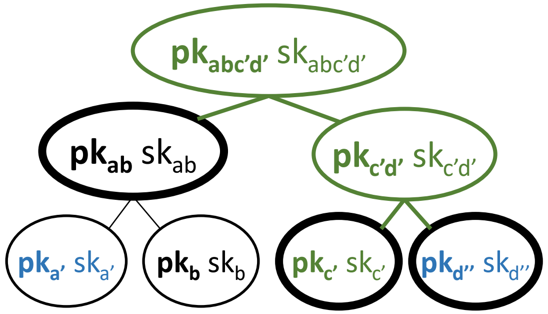
Agenda



PCS & ~~Concurrency~~ & Small overhead
~~PCS~~ & Concurrency & Small overhead
PCS & Concurrency & ~~Small overhead~~

Previous Work: What's the Problem? → **Lower Bound: What's the minimal overhead?**



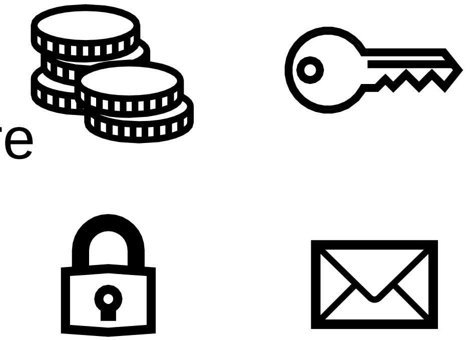


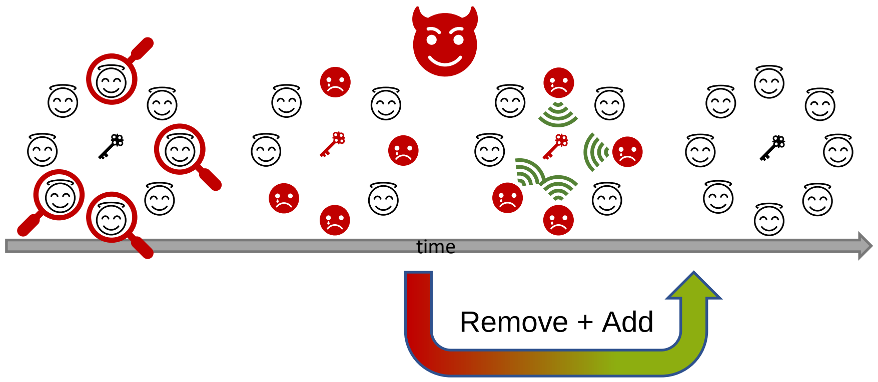
Upper Bound: Almost optimal construction → **Open Questions …**

$\Omega(t)$ vs. $O(t\cdot(1+\log(n/t)))$?
NIKE?
PCS-Delay?
Forward-secrecy?
Application to MLS

Lower Bound: What's the minimal overhead?

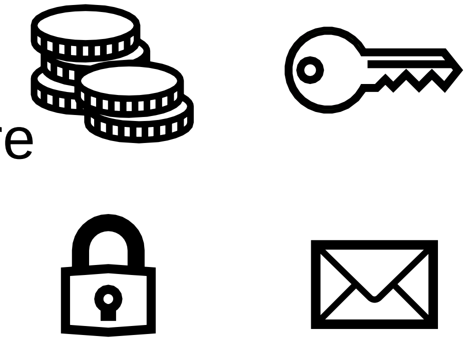
- Symbolic model
  - Variables are symbols without bit representation or algebraic structure
  - Algorithms follow "transition rules"
  - Round based execution

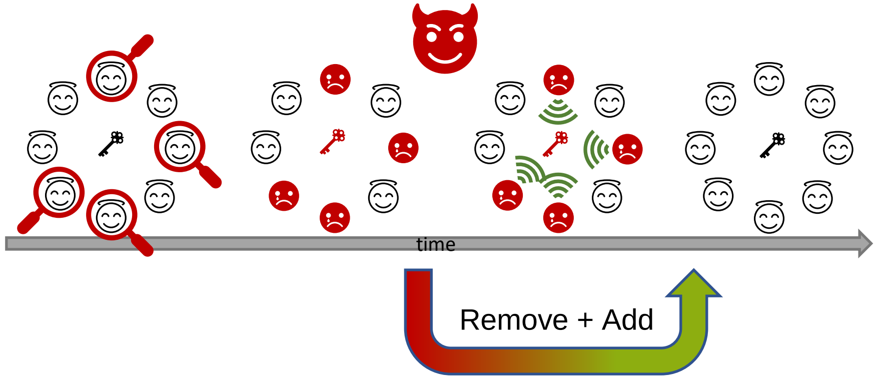




Lower Bound:
What's the minimal overhead?

- Symbolic model
  - Variables are symbols without bit representation or algebraic structure
  - Algorithms follow "transition rules"
  - Round based execution

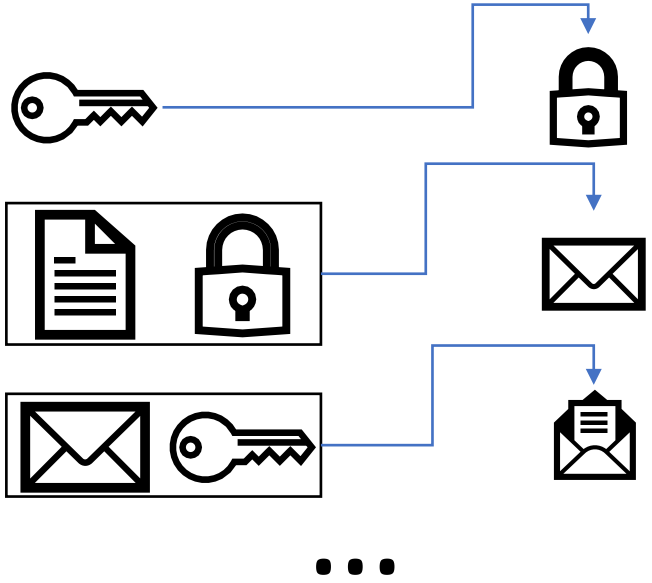




- Fixed set of allowed building blocks (for constructions with minimal overhead under PCS)

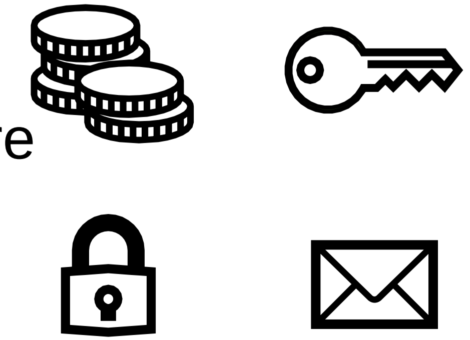
  Our "transition rules" model:
  - (Dual) pseudo-random functions
  - Key-updatable public key encryption (see [BRV20])
  - Broadcast encryption

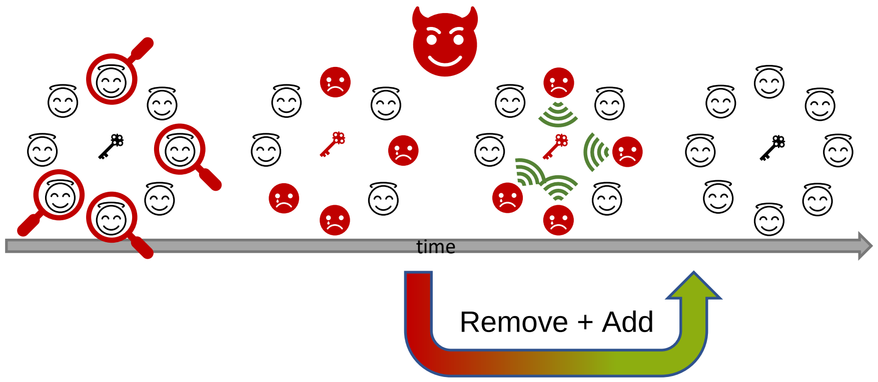
  → *More* than what previous constructions used



Lower Bound:
What's the minimal overhead?

- Symbolic model
  - Variables are symbols without bit representation or algebraic structure
  - Algorithms follow "transition rules"
  - Round based execution

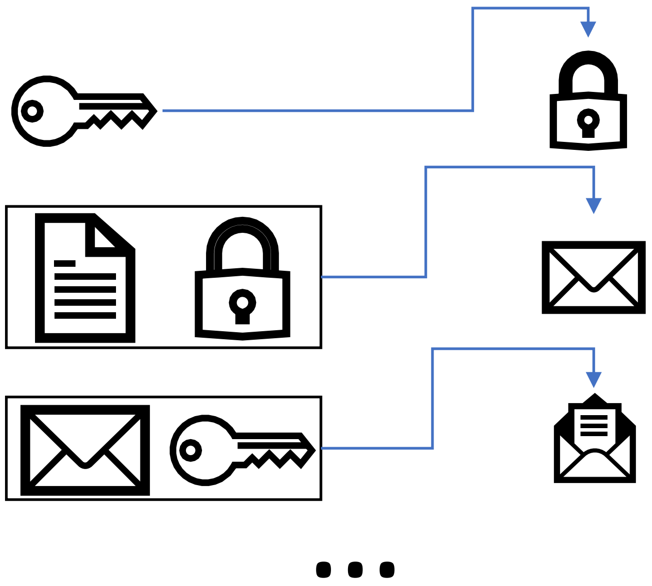




- Fixed set of allowed building blocks (for constructions with minimal overhead under PCS)

  Our "transition rules" model:
  - (Dual) pseudo-random functions
  - Key-updatable public key encryption (see [BRV20])
  - Broadcast encryption

  → *More* than what previous constructions used



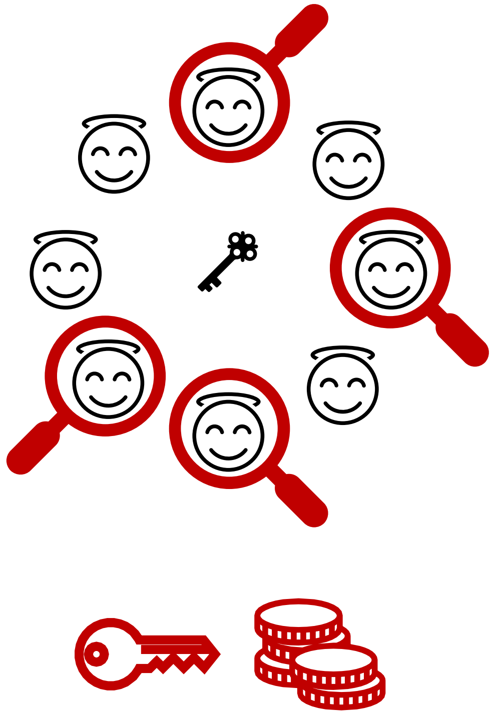
- Inspired by [MP04]:
  Lower bound O(log n) for forward-secure DGKE

Lower Bound: What's the minimal overhead?

Round i-2

i-2 Exposure:

- No (shared) secrets



Lower Bound: What's the minimal overhead?

i-2 Exposure:

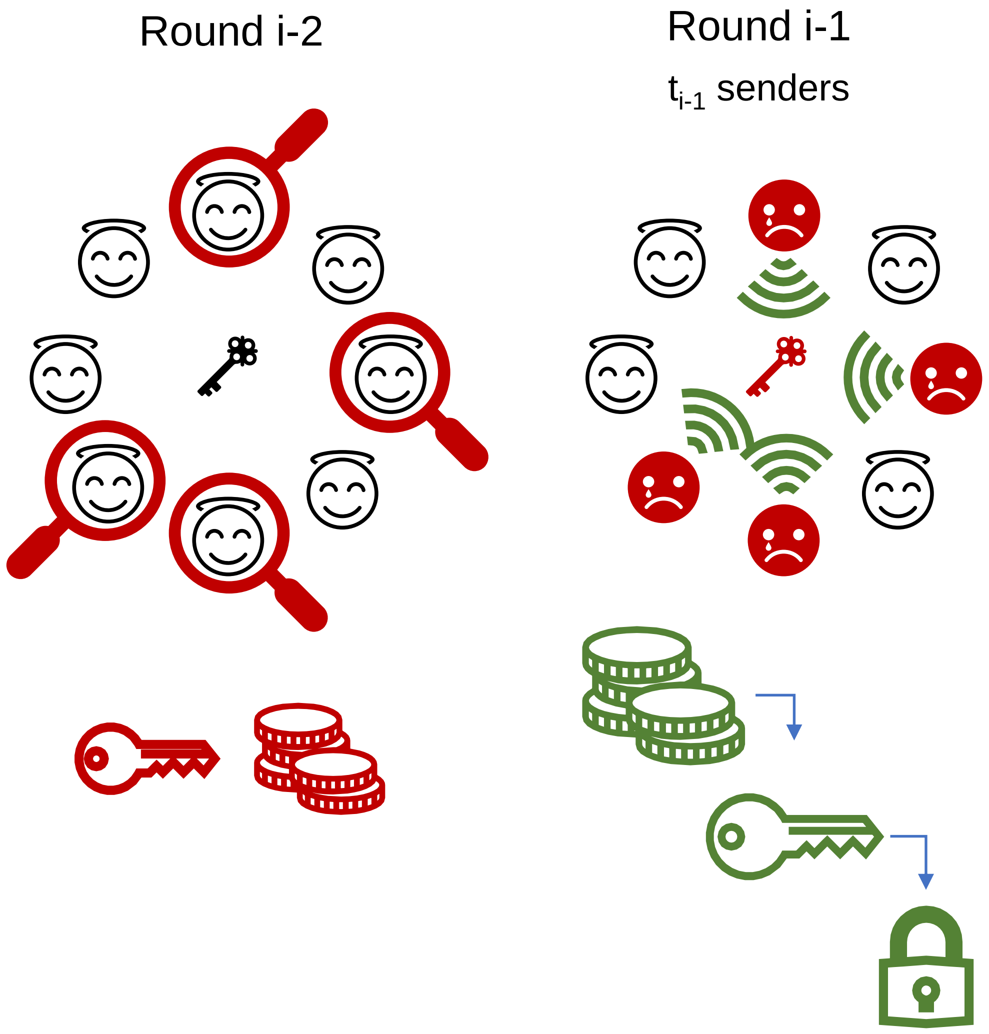
- No (shared) secrets

i-1 Recovery 1:

- Still no (shared) secrets
- Sampling of new secrets
- Sharing of derived values

→ Still no (shared) secrets

→ Though, public values of shared secrets



Lower Bound: What's the minimal overhead?

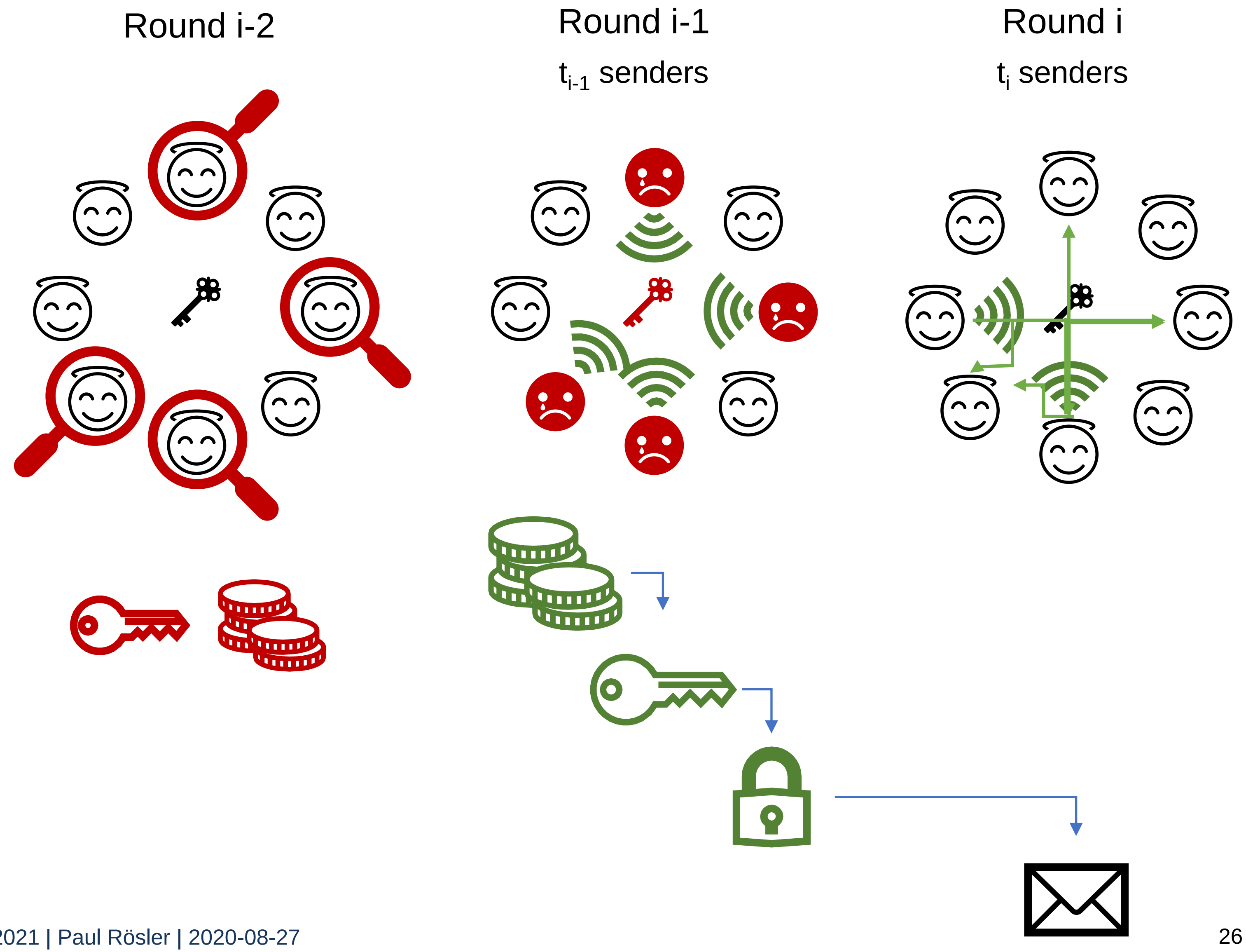
i-2 Exposure:
- No (shared) secrets

i-1 Recovery 1:
- Still no (shared) secrets
- Sampling of new secrets
- Sharing of derived values

→ Still no (shared) secrets

→ Though, public values of shared secrets

i Recovery 2:
- Respond to public values
- All senders must respond to every sender from i-1 as they cannot coordinate



Lower Bound:
What’s the minimal overhead?

i-2 Exposure:
- No (shared) secrets

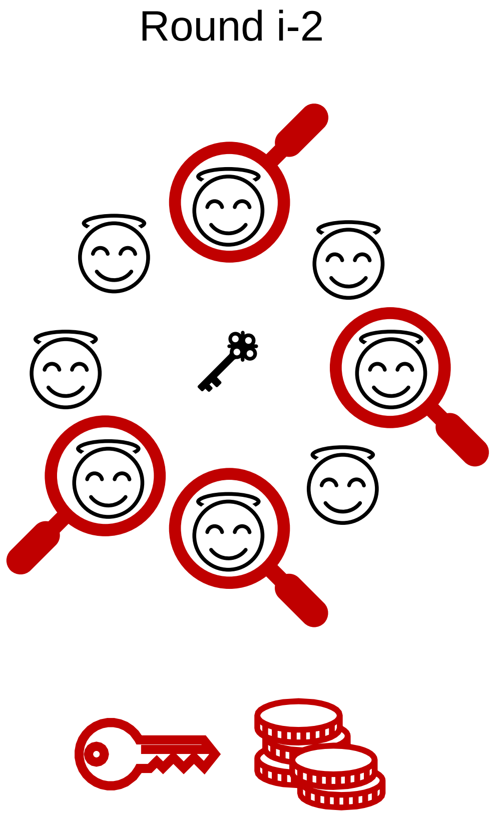
i-1 Recovery 1:
- Still no (shared) secrets
- Sampling of new secrets
- Sharing of derived values

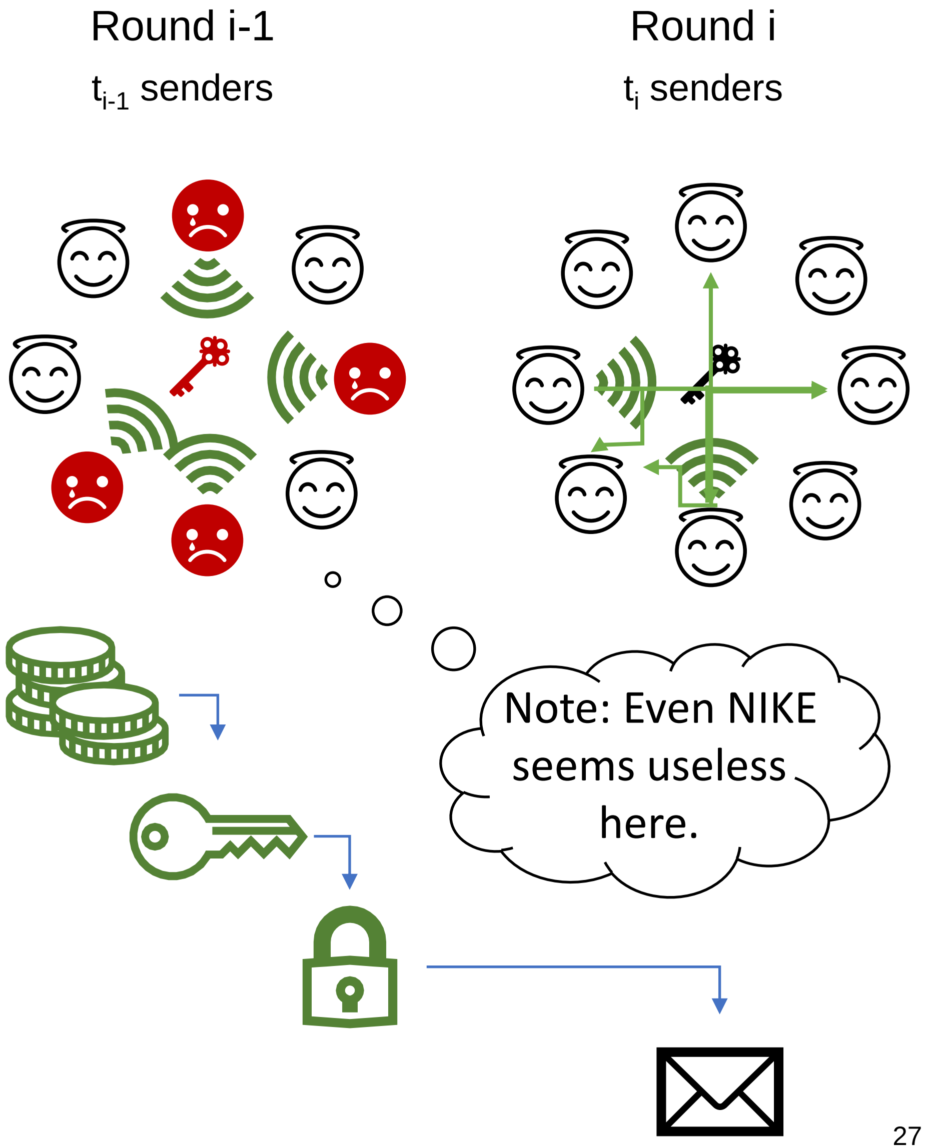
→ Still no (shared) secrets
→ Though, public values of shared secrets

i Recovery 2:
- Respond to public values
- All senders must respond to every sender from i-1 as they cannot coordinate

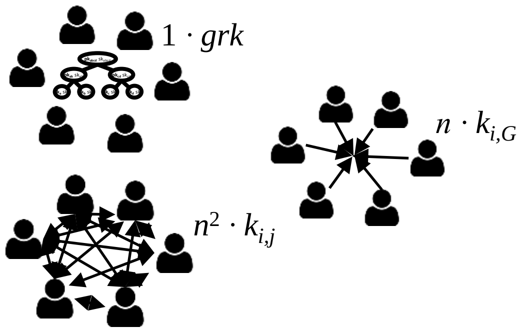
→ Each sender sends ≥ $(t_{i-1}-1)$ responses
→ ≥ $(t_{i-1}-1) \cdot t_i$ shares in round i

⇒ Overhead per recovery under t-concurrency: $\Omega(t)$

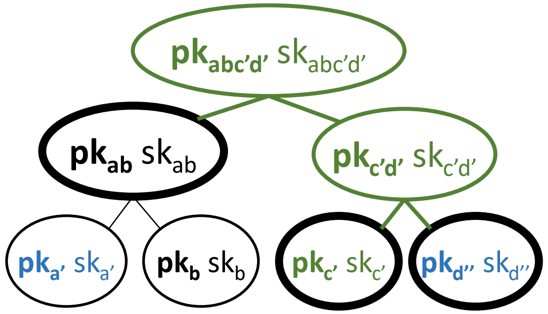




Agenda



PCS & ~~Concurrency~~ & Small overhead
~~PCS~~ & Concurrency & Small overhead
PCS & Concurrency & ~~Small overhead~~

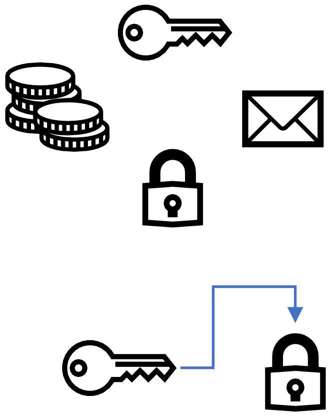


Previous Work: What's the Problem?

→ Lower Bound: What's the minimal overhead?

→ **Upper Bound: Almost optimal construction**

→ Open Questions …

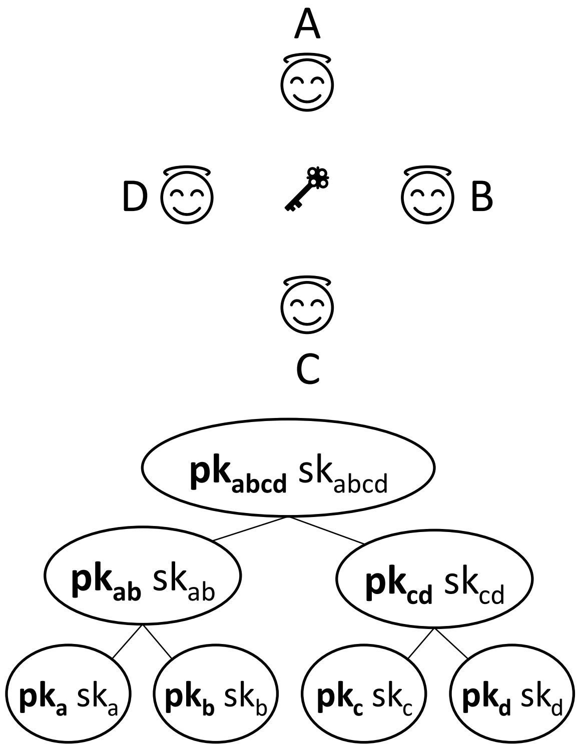


Realistic symbolic model:
No coordination + PCS
⇒ Ω(t)

Ω(t) vs. O(t·(1+log(n/t))?
NIKE?
PCS-Delay?
Forward-secrecy?
Application to MLS

Upper Bound: Almost optimal construction

Key tree (with updatable KEM)



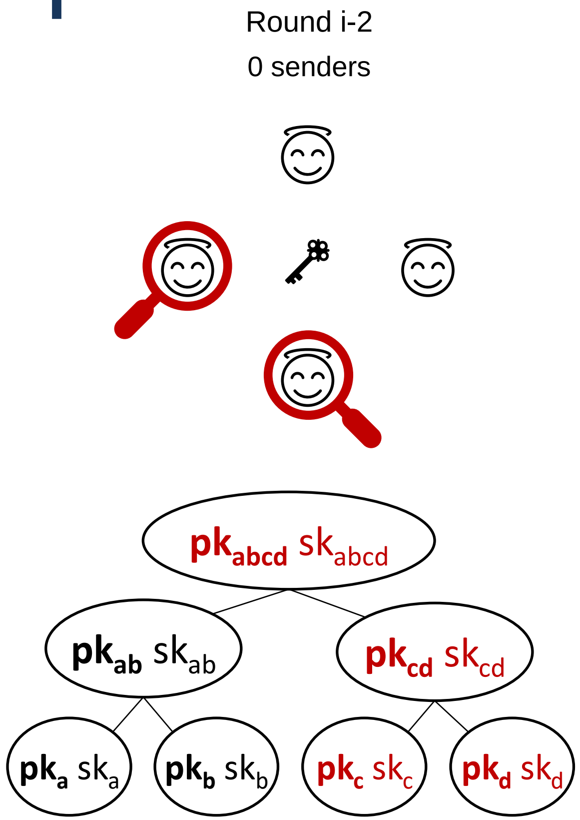
root secret
= group key

Upper Bound: Almost optimal construction

Key tree (with updatable KEM)

i-2 Exposure:

- Paths of c and d *public*: sk_c, sk_d, sk_{cd}, sk_{abcd}



Upper Bound: Almost optimal construction

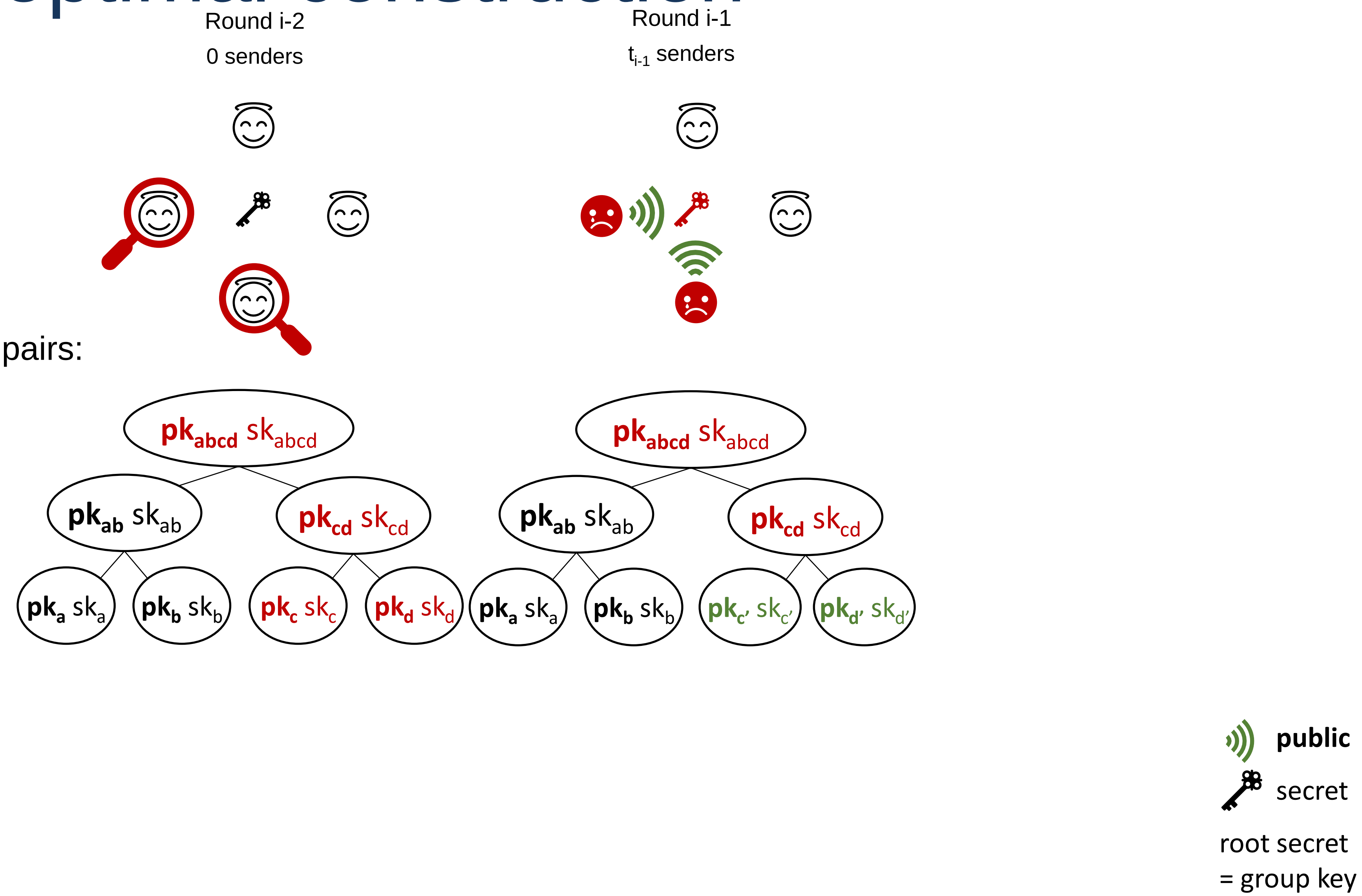
Key tree (with updatable KEM)

i-2 Exposure:

- Paths of c and d *public*: sk_c, sk_d, sk_{cd}, sk_{abcd}

i-1 Recovery 1:

- Generate and share new leaf key pairs: $(sk_{c'}, pk_{c'})$, $(sk_{d'}, pk_{d'})$



Upper Bound: Almost optimal construction

Key tree (with updatable KEM)

i-2 Exposure:

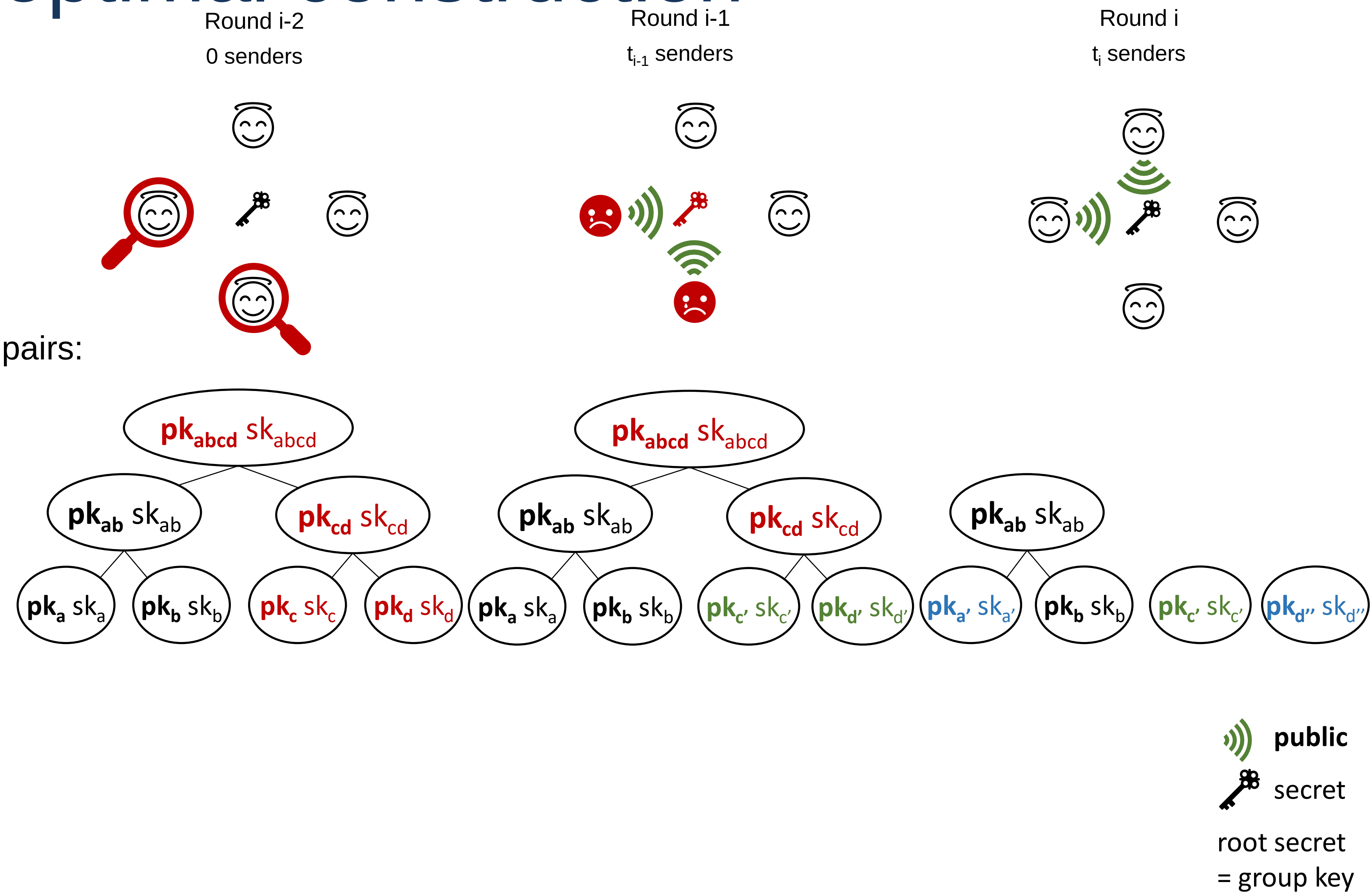
- Paths of c and d *public*: sk_c, sk_d, sk_{cd}, sk_{abcd}

i-1 Recovery 1:

- Generate and share new leaf key pairs: $(sk_{c'},pk_{c'})$, $(sk_{d'},pk_{d'})$

i Recovery 2:

a) See Recovery 1



Upper Bound: Almost optimal construction

Key tree (with updatable KEM)

i-2 Exposure:

- Paths of c and d *public*: sk_c, sk_d, sk_{cd}, sk_{abcd}

i-1 Recovery 1:

- Generate and share new leaf key pairs: $(sk_{c'}, pk_{c'})$, $(sk_{d'}, pk_{d'})$

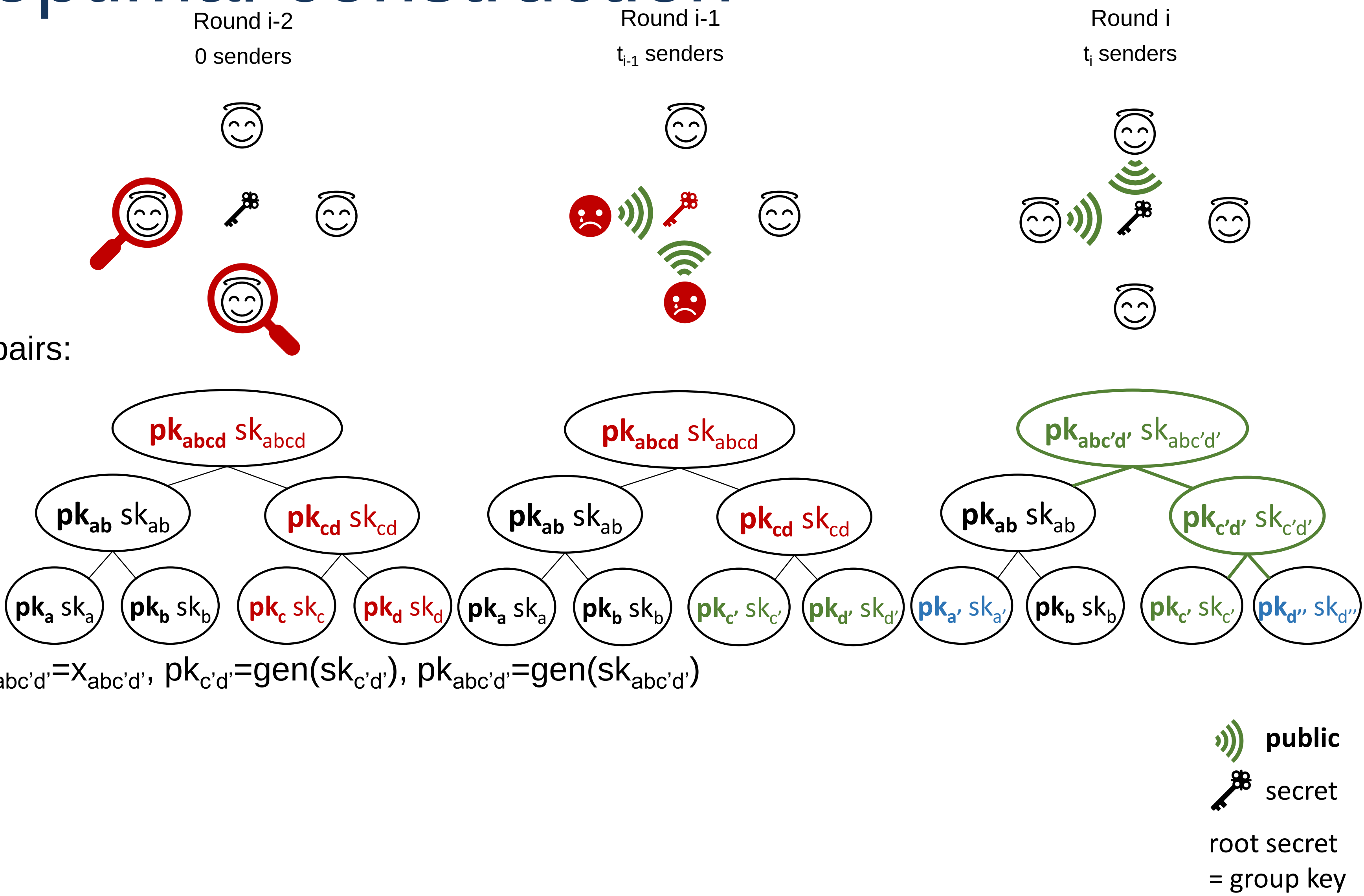
i Recovery 2:

a) See Recovery 1

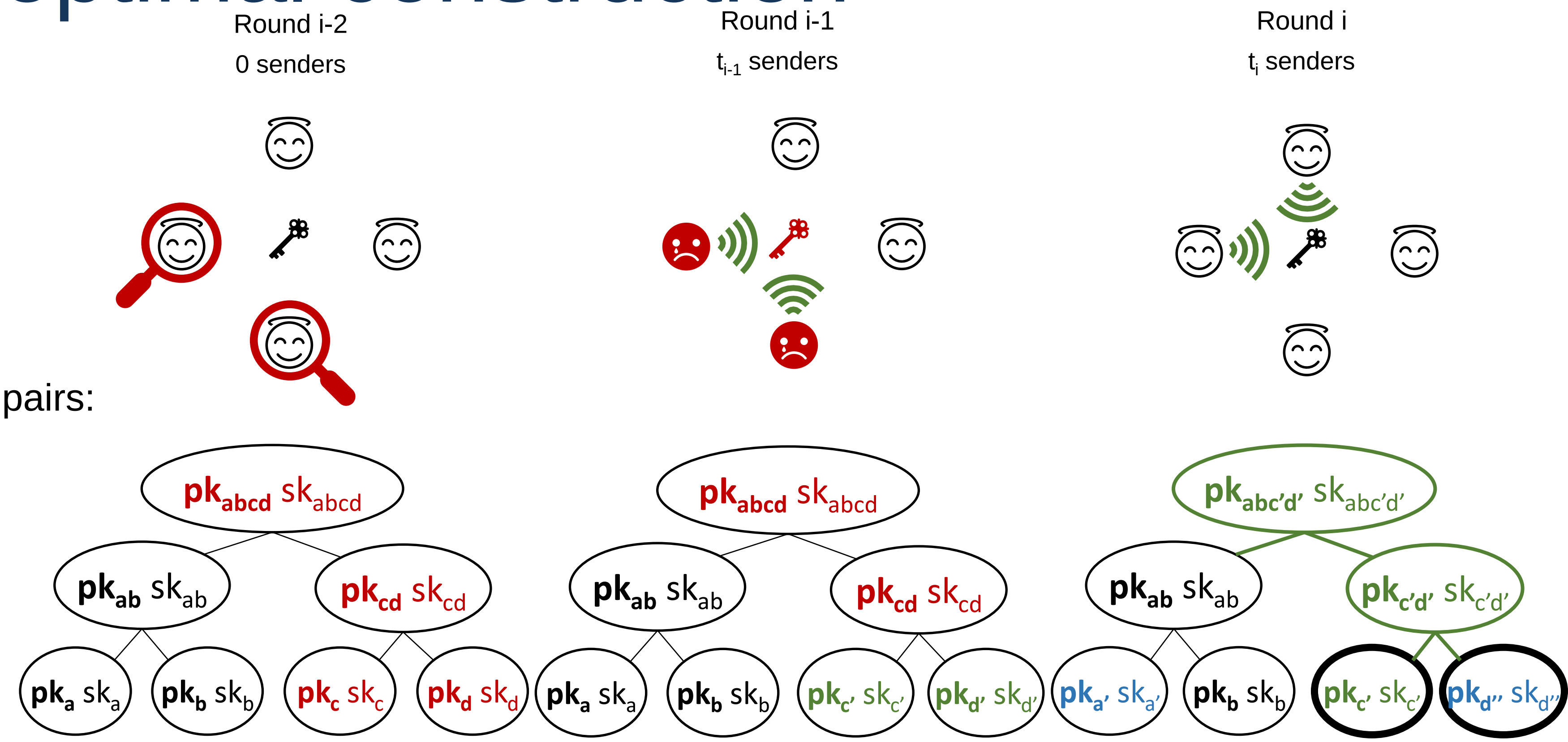
- Each sender generates new paths for previous senders:

b) Sample $x_{c'd'}$

c) Derive $sk_{c'd'} = x_{c'd'}$, $x_{abc'd'} = H(x_{c'd'})$, $sk_{abc'd'} = x_{abc'd'}$, $pk_{c'd'} = gen(sk_{c'd'})$, $pk_{abc'd'} = gen(sk_{abc'd'})$



Upper Bound: Almost optimal construction



Key tree (with updatable KEM)

i-2 Exposure:

- Paths of c and d *public*: sk_c, sk_d, sk_{cd}, sk_{abcd}

i-1 Recovery 1:

- Generate and share new leaf key pairs: $(sk_{c'},pk_{c'})$, $(sk_{d'},pk_{d'})$

i Recovery 2:

a) See Recovery 1

- Each sender generates new paths for previous senders:

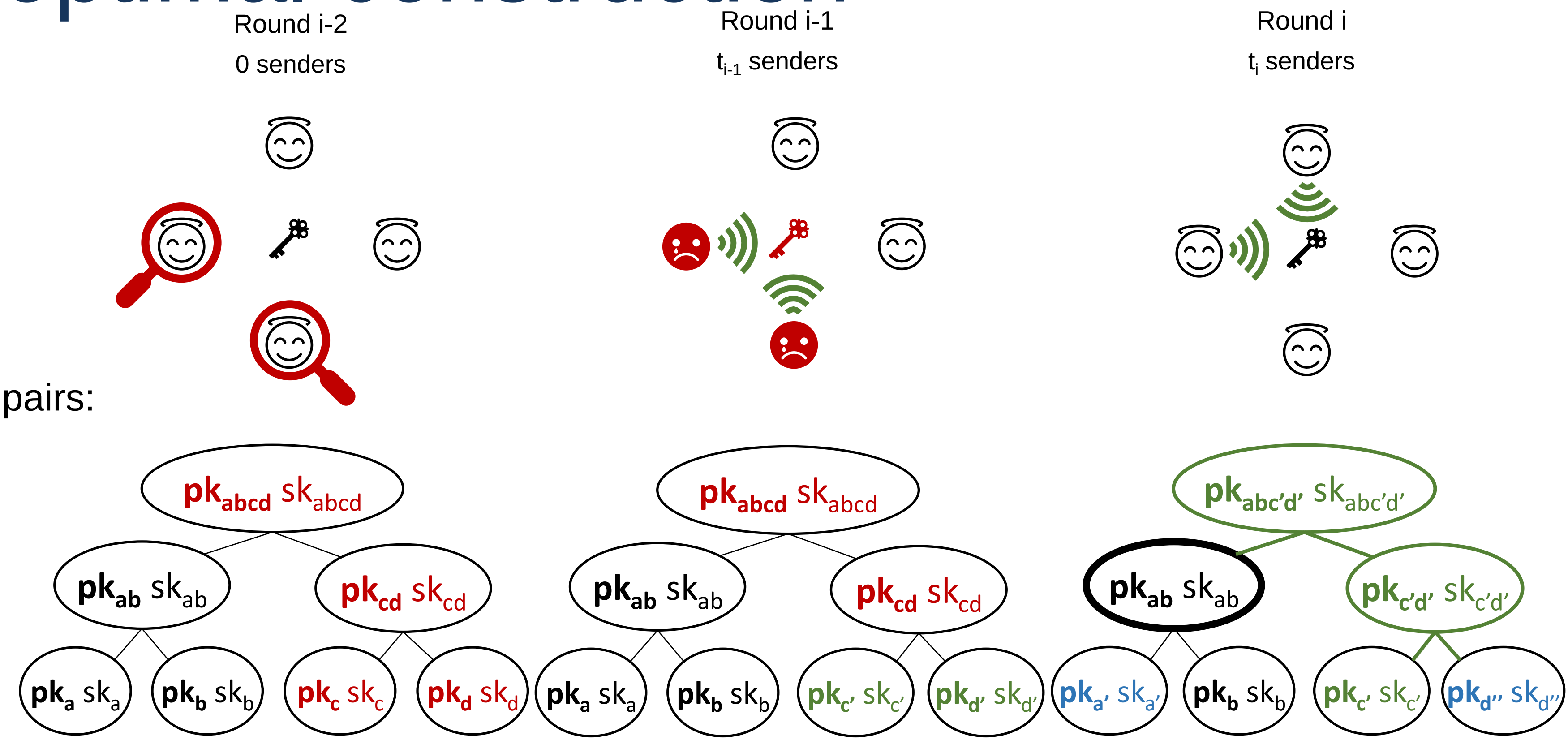
b) Sample $x_{c'd'}$

c) Derive $sk_{c'd'}=x_{c'd'}$, $x_{abc'd'}=H(x_{c'd'})$, $sk_{abc'd'}=x_{abc'd'}$, $pk_{c'd'}=gen(sk_{c'd'})$, $pk_{abc'd'}=gen(sk_{abc'd'})$

d) Send $enc(pk_{c'},x_{c'd'})$, $enc(pk_{d'},x_{c'd'})$

→ **Number of leafs: t_{i-1}**

Upper Bound: Almost optimal construction



Key tree (with updatable KEM)

i-2 Exposure:

- Paths of c and d *public*: sk_c, sk_d, sk_{cd}, sk_{abcd}

i-1 Recovery 1:

- Generate and share new leaf key pairs: $(sk_{c'},pk_{c'})$, $(sk_{d'},pk_{d'})$

i Recovery 2:

a) See Recovery 1

- Each sender generates new paths for previous senders:

b) Sample $x_{c'd'}$

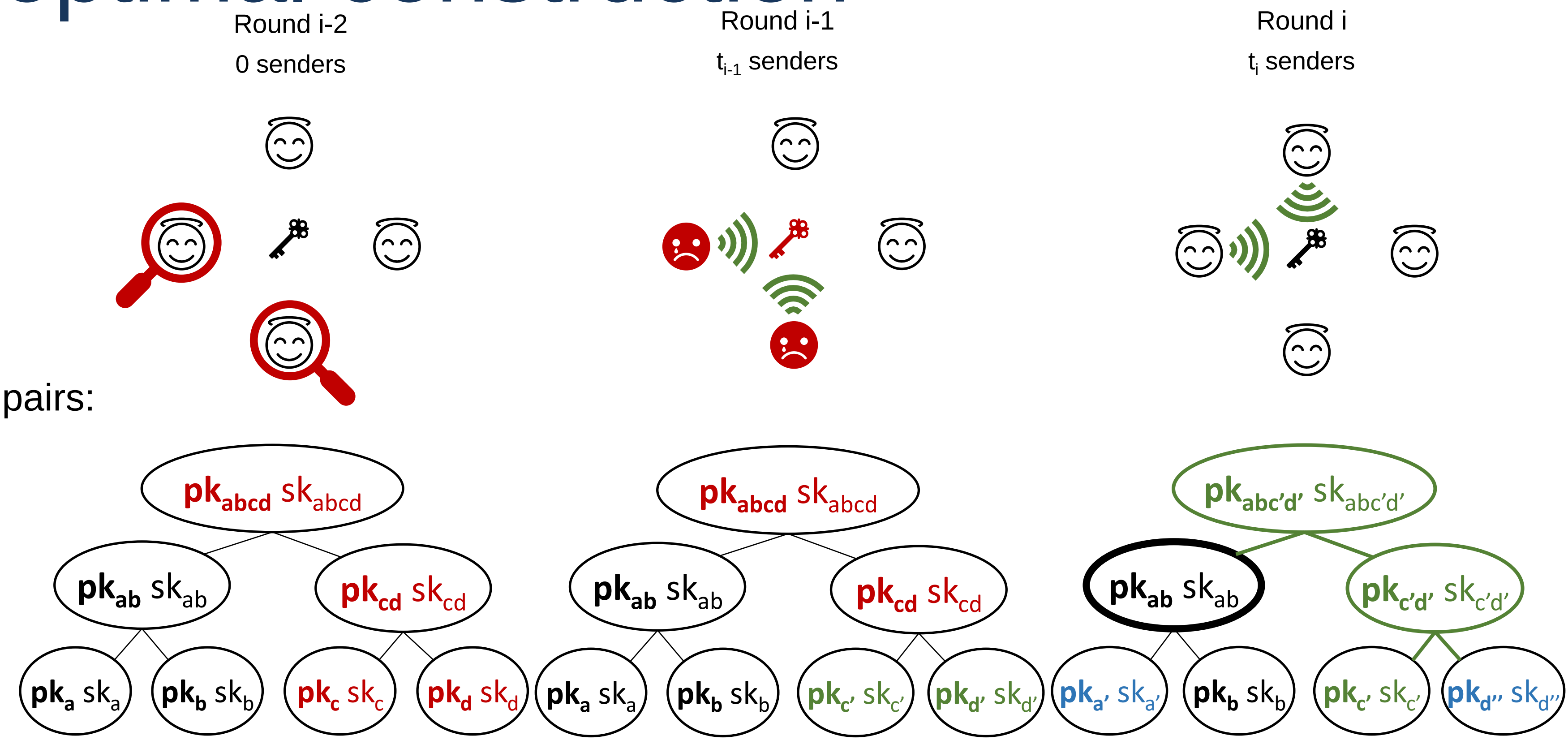
c) Derive $sk_{c'd'}=x_{c'd'}$, $x_{abc'd'}=H(x_{c'd'})$, $sk_{abc'd'}=x_{abc'd'}$, $pk_{c'd'}=gen(sk_{c'd'})$, $pk_{abc'd'}=gen(sk_{abc'd'})$

d) Send $enc(pk_{c'},x_{c'd'})$, $enc(pk_{d'},x_{c'd'})$, $pk_{c'd'}$, $enc(pk_{ab},x_{abc'd'})$

→ Number of leafs: t_{i-1}, **number of update-tree-siblings: $O(t_{i-1} \cdot \log(n/t_{i-1}))$**

public
secret
root secret = group key

Upper Bound: Almost optimal construction



Key tree (with updatable KEM)

i-2 Exposure:

- Paths of c and d *public*: sk_c, sk_d, sk_{cd}, sk_{abcd}

i-1 Recovery 1:

- Generate and share new leaf key pairs: $(sk_{c'},pk_{c'})$, $(sk_{d'},pk_{d'})$

i Recovery 2:

a) See Recovery 1

- Each sender generates new paths for previous senders:

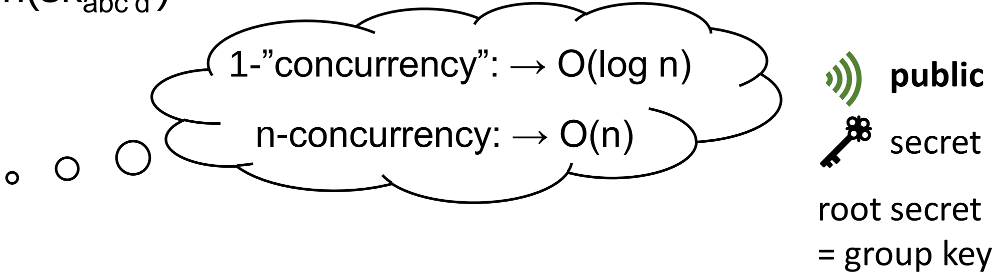
b) Sample $x_{c'd'}$

c) Derive $sk_{c'd'}=x_{c'd'}$, $x_{abc'd'}=H(x_{c'd'})$, $sk_{abc'd'}=x_{abc'd'}$, $pk_{c'd'}=gen(sk_{c'd'})$, $pk_{abc'd'}=gen(sk_{abc'd'})$

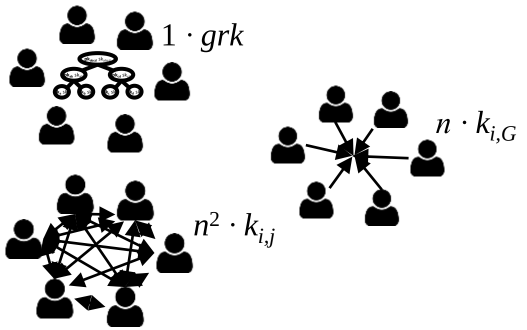
d) Send $enc(pk_{c'},x_{c'd'})$, $enc(pk_{d'},x_{c'd'})$, $pk_{c'd'}$, $enc(pk_{ab},x_{abc'd'})$

→ Number of leafs: t_{i-1}, **number of update-tree-siblings: $O(t_{i-1} \cdot \log(n/t_{i-1}))$**

⇒ Overhead per recovery under t-concurrency: $O(t+t \cdot \log(n/t))$

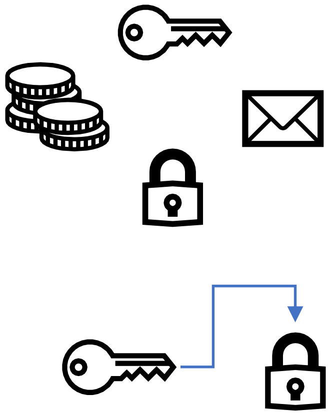


Agenda

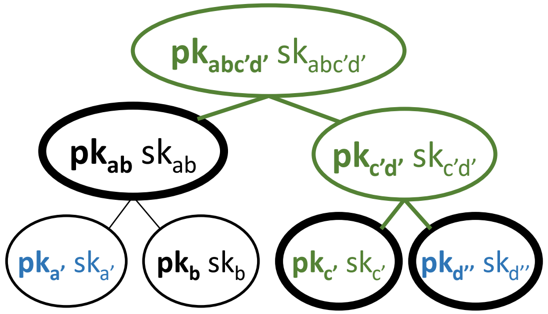


PCS & ~~Concurrency~~ & Small overhead
~~PCS~~ & Concurrency & Small overhead
PCS & Concurrency & ~~Small overhead~~

Previous Work: What's the Problem? → **Lower Bound: What's the minimal overhead?** → **Upper Bound: Almost optimal construction** → **Open Questions …**



Realistic symbolic model:
No coordination + PCS
$\Rightarrow \Omega(t)$



Two-step recovery
$\Rightarrow O(t \cdot (1+\log(n/t)))$

$\Omega(t)$ vs. $O(t \cdot (1+\log(n/t)))$?
NIKE?
PCS-Delay?
Forward-secrecy?
Application to MLS

@roeslpa Full details & formal proofs online *soonish*