SMMP: A Secure, Multi-party Messaging Protocol

David R. Andersen

Electrical and Computer Engineering
The University of Iowa
Iowa City IA 52242

k0rx@uiowa.edu

Mark S. Andersland

Electrical and Computer Engineering

The University of Iowa

Tycho J. Andersen

Canonical Inc.

Madison Wisconsin

Outline

- Motivation
- Definitions
- Threat model
- Review of one-on-one messaging protocols
- Review of multi-party messaging protocols
- SMMP – required cryptographic primitives
- SMMP – key agreement
- SMMP – send/receive
- SMMP – resync/join/depart
- Security analysis
- Efficiency analysis
- Demonstration: ratchet visualization and multi-party chat client

Motivation

- Mimic private group conversation with an online tool
  - Confidentiality – in a physical conversation, there is no long-lived record beyond the memory of the participants
  - Authentication – in a physical conversation, participants can authenticate each other face-to-face as well as authenticate the communicator of a message
  - Repudiation – during or following a physical conversation, participants can deny having made a statement or taken part to a 3rd party (deniability)
- Achieving these goals is not possible in a plaintext network – logs, transcripts, persistent bulk surveillance, ...
- As a result, online privacy concerns have become paramount – NSA, GCHQ, hackers, organized crime, ...

Definitions

- Perfect forward secrecy (PFS): compromise of a message key does not compromise previous keys
- Perfect future secrecy (PFuS): compromise of a message key does not compromise future keys
- Plausible deniability (PD): the ability of a participant in a conversation to plausibly deny having taken part in the conversation – no algorithmic proof of participation
- Ratchet: a mechanism for deterministically changing symmetric encryption keys on a per-message basis that provides both PFS and PFuS

Threat Model

We use a generalized mpOTR threat model with the following players (after Goldberg, et.al. 2009).

- Security adversary – The goal of this entity is to read messages he is not entitled to.
  - Global passive adversary
  - Cannot ask for participant private information until after the conversation is over (so-called rubber hose cryptography)
  - Cannot participate in the conversation or control those who do
- Consensus adversary – The goal of this entity is to get an honest user Alice to reach consensus with another honest user Bob on a transcript where the transcripts do not actually match.
  - Can control dishonest participants to make conflicting statements to different participants
  - Can drop, duplicate, reorder, and replay messages – these actions may or may not be useful depending on the protocol
- Privacy adversary – the goal of this entity is, given a (real or fake) transcript of a conversation, to convince a judge (**online** or offline) that Alice took part in the conversation, *i.e.* read messages in the conversation, or authored messages in the conversation.
  - Can participate in the conversation
  - Offline judges are only given a transcript after the conversation is over
  - Online judges can also participate in the conversation

I. Goldberg, M. D. Van Gundy, B. Ustanoglu, and H. Chen, *CSS'09*, pp. 358-368, ACM (2009).

Adversary Capabilities

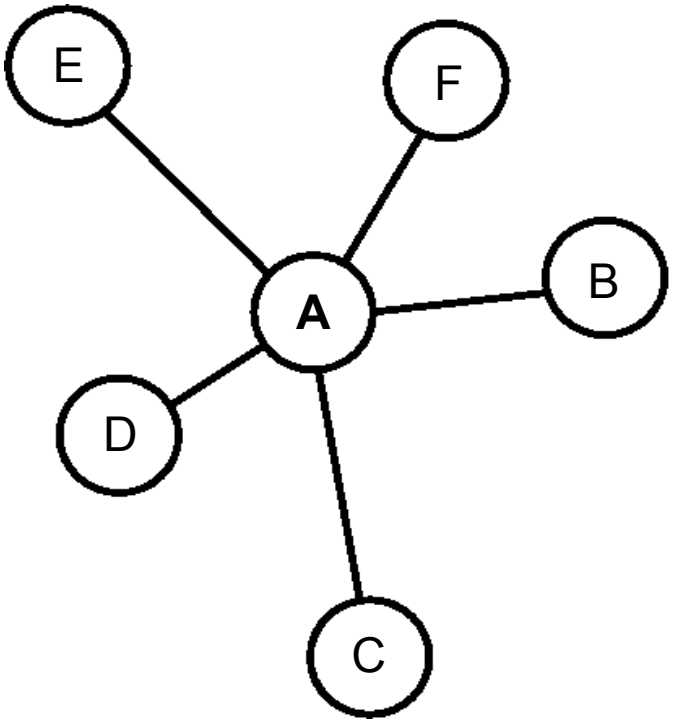
- Security adversary remains passive until the group conversation is complete. He may then ask participants for their private keys.
- Consensus and privacy adversaries may participate in the conversation as dishonest participants or control those who do.
  - Consensus and privacy adversaries have access to a complete (plaintext and ciphertext) transcript of the conversation.
- Consensus and privacy adversaries may or may not follow the protocol faithfully in order not to reveal their adversarial nature, but may ask honest users to send particular messages as part of the conversation.
  - Consensus and privacy adversaries who do not follow the protocol faithfully will ultimately be removed and the group will re-form without them.
  - Protocol should detect bad actors (those who do not follow the protocol faithfully).

Review of One-on-One Messaging Protocols

- OTR
  N. Borisov, I. Goldberg, and E. Brewer, *WPES'04*, ACM (2004).
  M. D. Raimondo, R. Gennaro, and H. Krawczyk, WPES'05, ACM (2005).
  - *Advertise* → *Acknowledge* → *Use* ephemeral rekeying
    - Multiple messages compromised if one ephemeral key is compromised (window of compromise)
  - Requires publishing ephemeral keys for PD
  - Self-healing
- SCIMP
  V. Moscaritolo, G. Belvin, and P. Zimmerman, http://silentcircle.com/static/download/SCIMP%20paper.pdf
  - newKey = hash(oldKey) rekeying
  - No PFuS
- Axolotl
  T. Perrin and M. Marlinspike, https://github.com/trevp/axolotol/wiki/newversion
  - Hybrid rekeying – combination of hash() iteration and new ephemeral key
    - Asynchronous – allows for out-of-order message receipt with reduction in PFS
  - Self-healing

Review of multi-party messaging protocols

- Group OTR
  - virtual server must be assumed honest: decrypt/reencrypt
  - leaks conversation metadata
  - no transcript verification
  - no authentication of participants except to server



J. Bian, R. Seker, and U. Topaloglu, *IRI'07*, IEEE Comp. Soc. (2007).

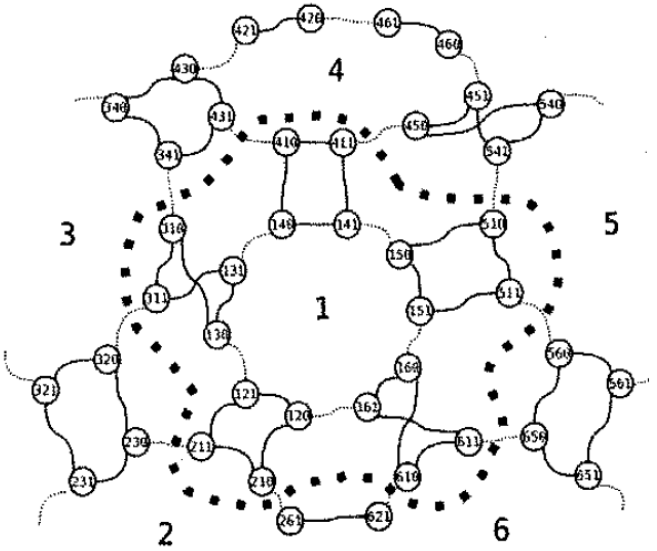
multi-party OTR

- PD against offline judges only
- Links all messages together – if you commit to one part of a conversation, you've committed to it all
- Requires more than one party to forge a transcript – affects PD
- Allows a “window of compromise” - one key compromise allows attacker to read several messages in the chain
- Requires broadcasting ephemeral keys at end of the conversation for PD

I. Goldberg, M. D. Van Gundy, B. Ustanoglu, and H. Chen, *CSS'09*, ACM (2009).

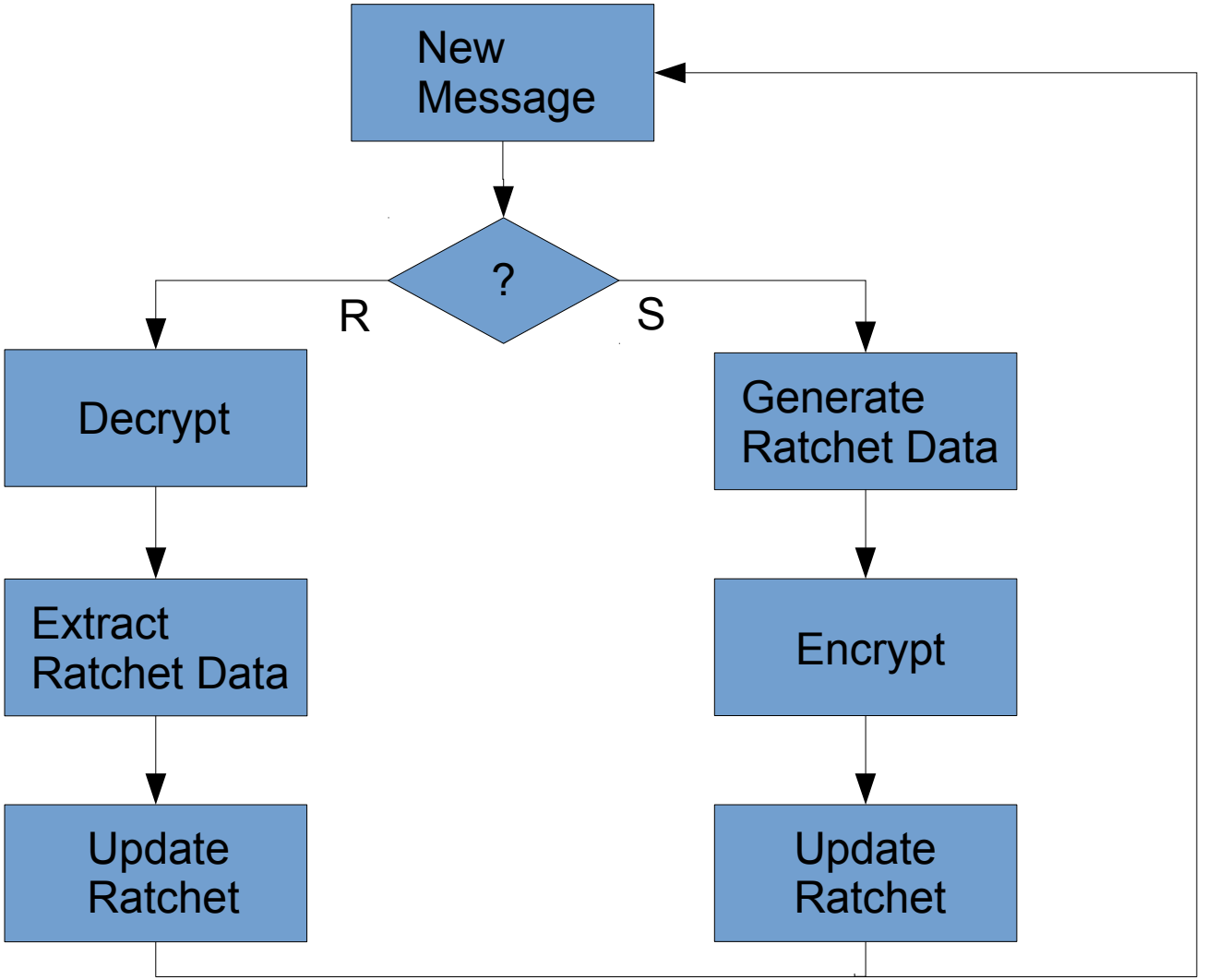
Improved GOTR

- Complexity – 6 round GKA required each time ephemeral group key is updated (no PFS without this!)
- Complexity – each participant becomes 2(N-1) nodes for 2-layer group key agreement
- Underlying binary secure channels between users are required
- Requires broadcasting of ephemeral keys for PD



H. Liu, E. Y. Vasserman, and N. Hopper, *WPES'13*, ACM (2013).

High-level Overview of SMMP

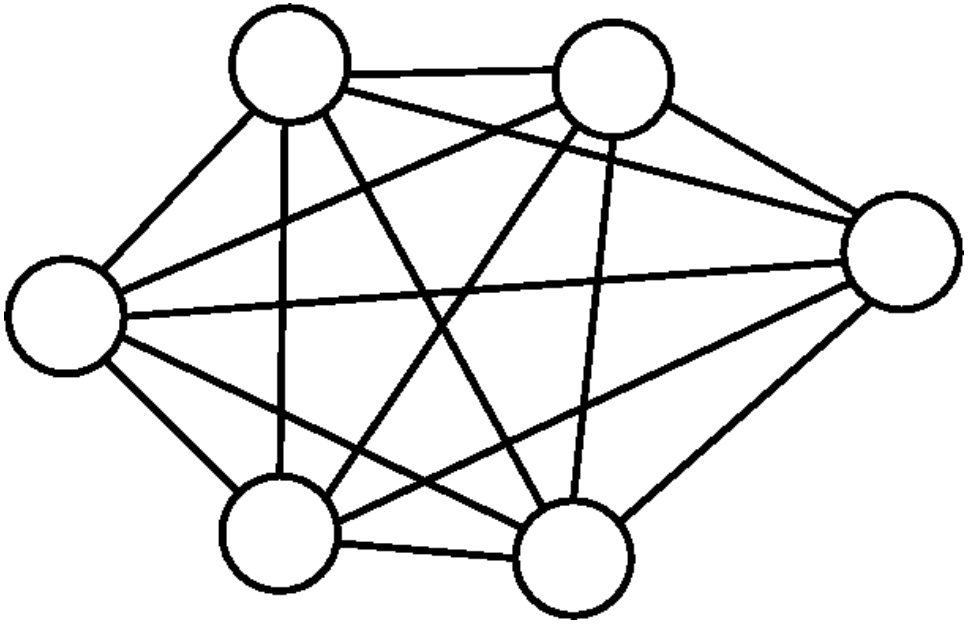


SMMP – Required cryptographic primitives

- Underlying secure symmetric encryption algorithm $c=e(k,m)$, $m=d(k,c)$ (*e.g.* AES256)
- ECDH on some elliptic curve (*e.g.* curve25519)
- DH on some finite field (*e.g.* 2048 bit MODP prime from RFC 3526)
- Secure cryptographic hash function (*e.g.* SHA256)
- Secure cryptographic message authentication code (*e.g.* HMAC)
- Secure key definition function (*e.g.* PBKDF2)

SMMP – Key Agreement

- Requirements
  - Peer-to-peer
  - Each peer should independently authenticate every other peer
  - No security requirements should be imposed on the underlying transport layer (information exchange in plaintext)



Burmester-Desmedt[1] group key agreement takes place over a fully-connected network

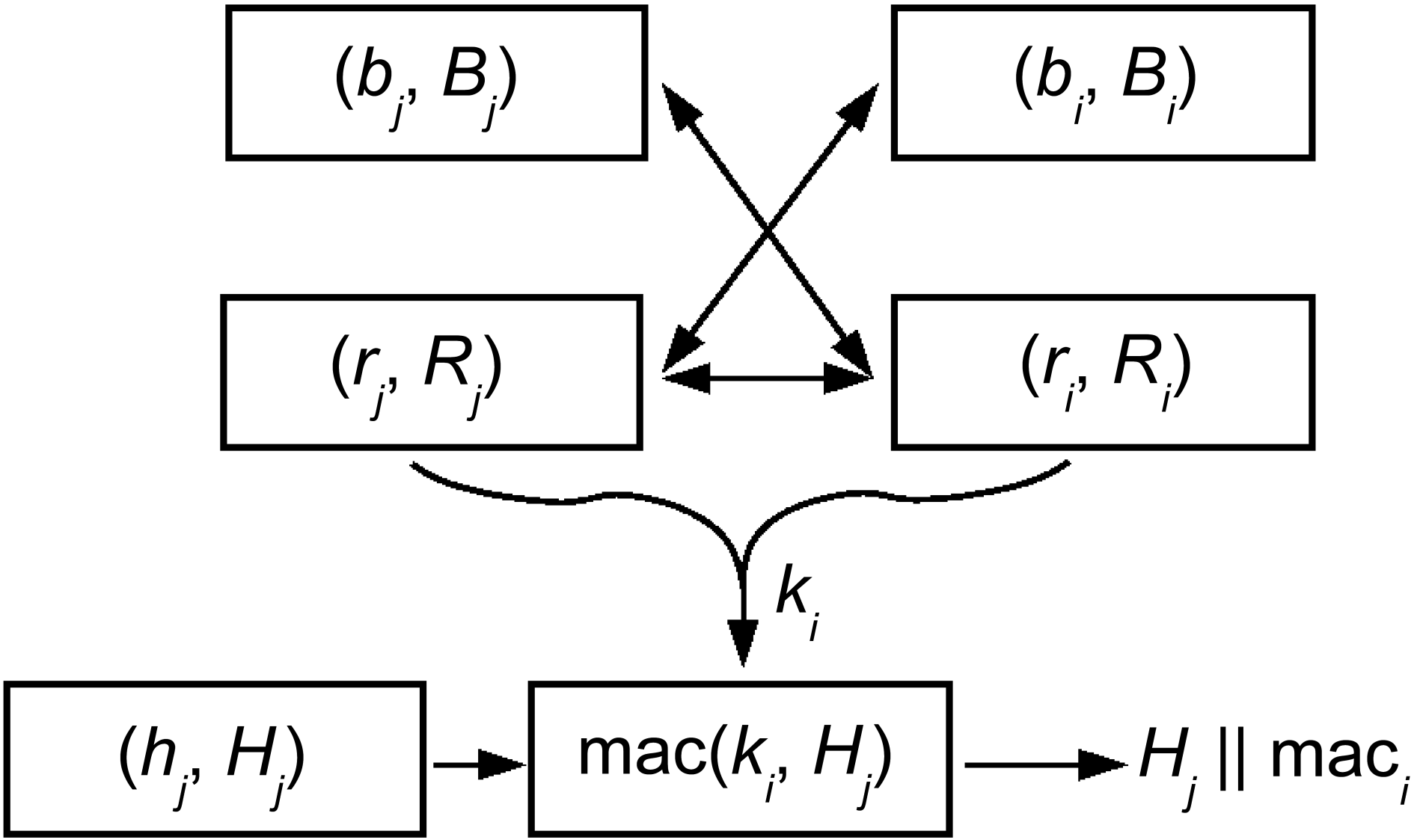
Identity is proved by knowledge of a private key corresponding to a known public key

Authentication is provided via a triple DH Mechanism – a variant of Just and Vaudenay[2]

[1]M. Burmester and Y. Desmedt, *EUROCRYPT'94*, LN in CS v. 950 (1995).

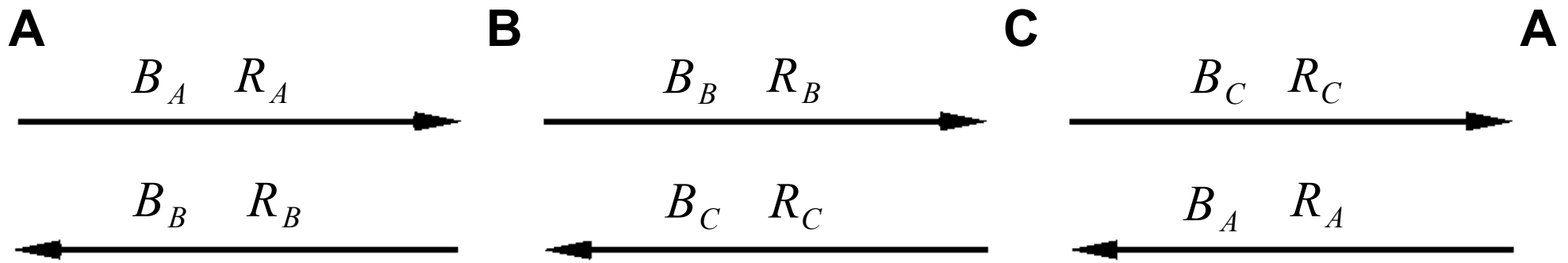
[2]M. Just and S. Vaudenay, ASIACRYPT'96 (1996).

Authentication Step – P_j to P_i



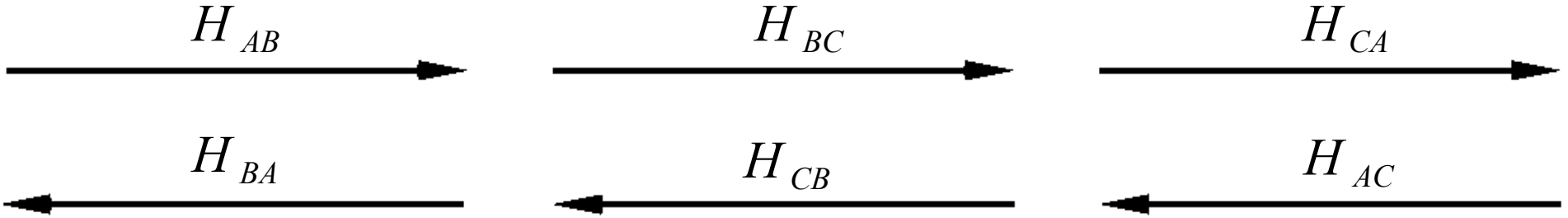
Authenticated BD Group Key Exchange

Each participant generates: $(\boldsymbol{b_j}, \boldsymbol{B_j}), (r_j, R_j), (h_j, H_j)$

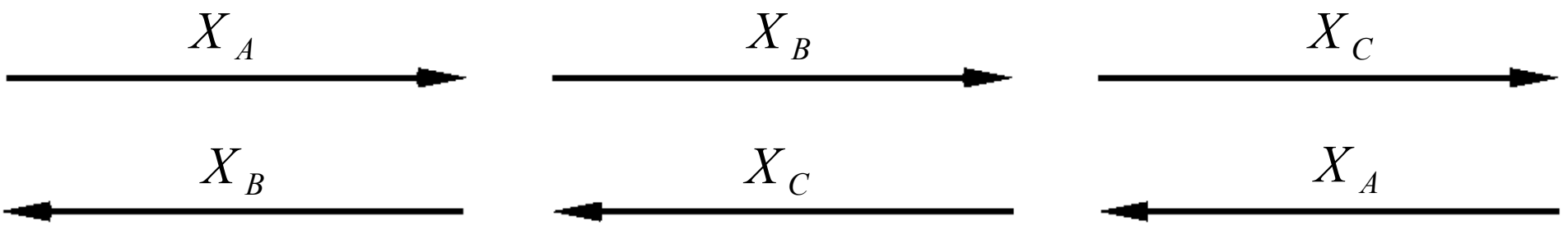


Compute: $k_{ji} = \text{hash}(\text{ECDH}(b_j, R_i) \,\|\, \text{ECDH}(r_j, B_i) \,\|\, \text{ECDH}(r_j, R_i))$

$H_{ji} = H_j \,\|\, \text{mac}(k_{ji}, H_j)$



Compute: $X_j = BD_1(h_j, \{H_i\})$



Compute: $\text{MK} = BD_2(h_j, \{X_i\})$

Key Agreement steps

- Preliminary Steps
  - The total number of participants *N* is distributed to all putative participants.
  - Diffie-Hellman parameters for both ECDH and DH protocols are agreed upon by all participants.
  - A unique participant index *i* is assigned to each participant [known by persistent identity key (b_i, B_i)]. The method of assigning this index is application-dependent and may *e.g.* be assigned based on the ordering of the participant public identity keys.

Key Agreement continued

- Each participant P_j completes the following steps:
  - Each participant P_j generates an ephemeral ECDH ratchet key pair (r_j, R_j) and an ephemeral finite-field DH handshake key pair (h_j, H_j).
  - Participant P_j broadcasts B_j and R_j.
  - Participant P_j computes the set of keys:

$$k_{TDHi} = \text{hash}\left(\text{ECDH}\left(b_j, R_i\right) \| \text{ECDH}\left(r_j, B_i\right) \| \text{ECDH}\left(r_j, R_i\right)\right)$$

    and macs:

$$\text{mac}_i = \text{mac}\left(k_{TDHi}, H_j\right)$$

Key Agreement continued

- P_j sends H_j || mac_i to participant P_i for all participants.
- P_j receives H_i || mac_j from each participant P_i.
- P_j confirms the mac for each handshake key, thereby authenticating the identity of each P_i.
- P_j Computes $K_j = (H_{j+1}/H_{j-1})^{h_j} \bmod p$. P_j broadcasts K_j to all participants P_i.
- P_j receives K_i from each participant P_i and computes the master key:

$$\mathrm{MK} = \mathrm{hash}\left(H_{j-1}^{N h_j} K_j^{N-1} K_{j+1}^{N-2} \ldots K_{j-2} \bmod p \right)$$

Initial Ratchet State

Root Key:

$$\mathrm{RK} = \mathrm{KDF}\left(\mathrm{MK}, 0\mathrm{x}00\right)$$

Message Key:

$$\mathrm{mk} = \mathrm{KDF}\left(\mathrm{MK}, 0\mathrm{x}01\right)$$

Group Private Key:

$$v = \mathrm{KDF}\left(\mathrm{MK}, 0\mathrm{x}02\right)$$

Digest:

$$\mathrm{conv\_digest} = 0\mathrm{x}00 * 32$$

Resync Flag:

$$\mathrm{resync\_required} = False$$

Ratchet Keys:

$$\left\{R_i\right\} \text{ from participants}$$

Private Ratchet Key:

$$r_j \text{ from key agreement}$$

Initialization Ratchet Keys:

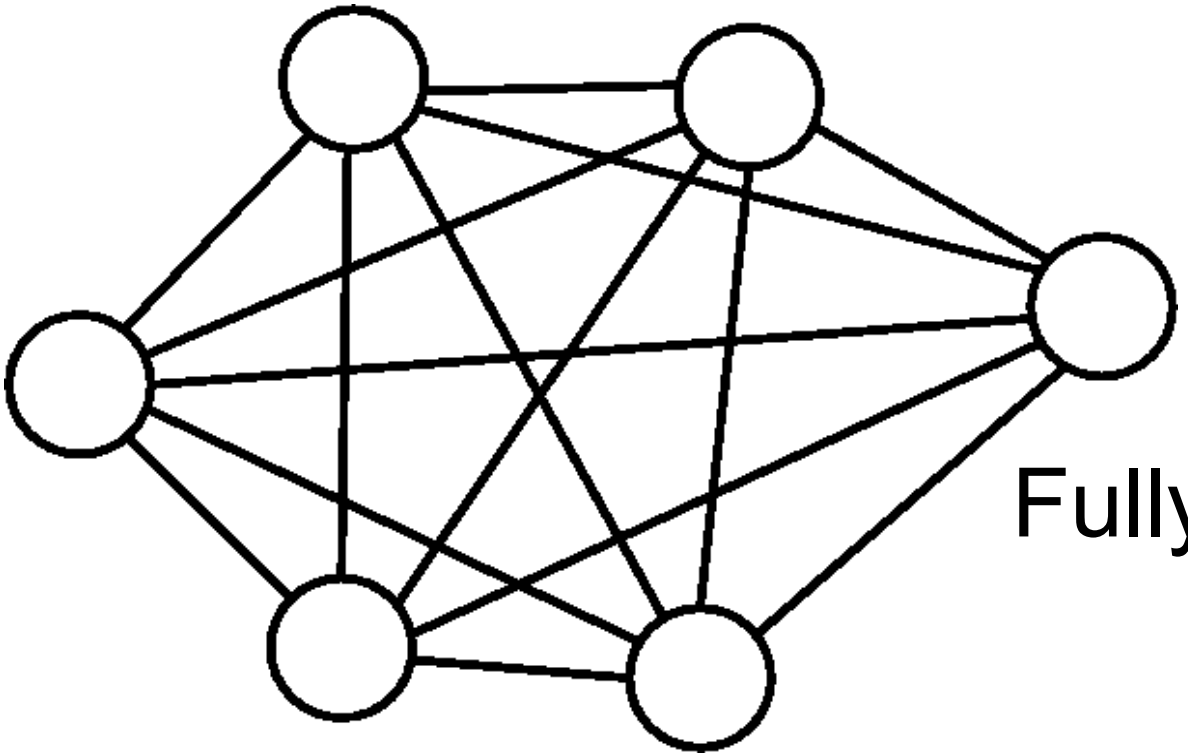
$$\left\{R_i^{init}\right\} \text{ from participants}$$

Initialization Private Ratchet Key:

$$r_j^{init} \text{ from key agreement}$$

Group Parameters:

$$N,\ j \text{ from key agreement}$$

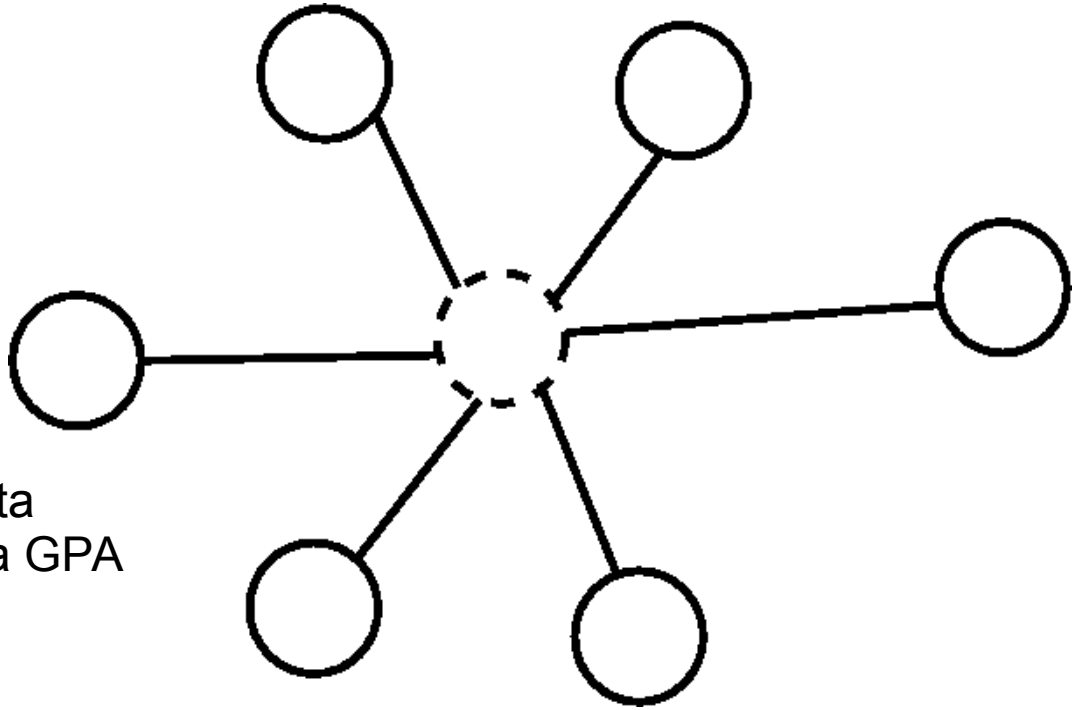


Fully-connected graph

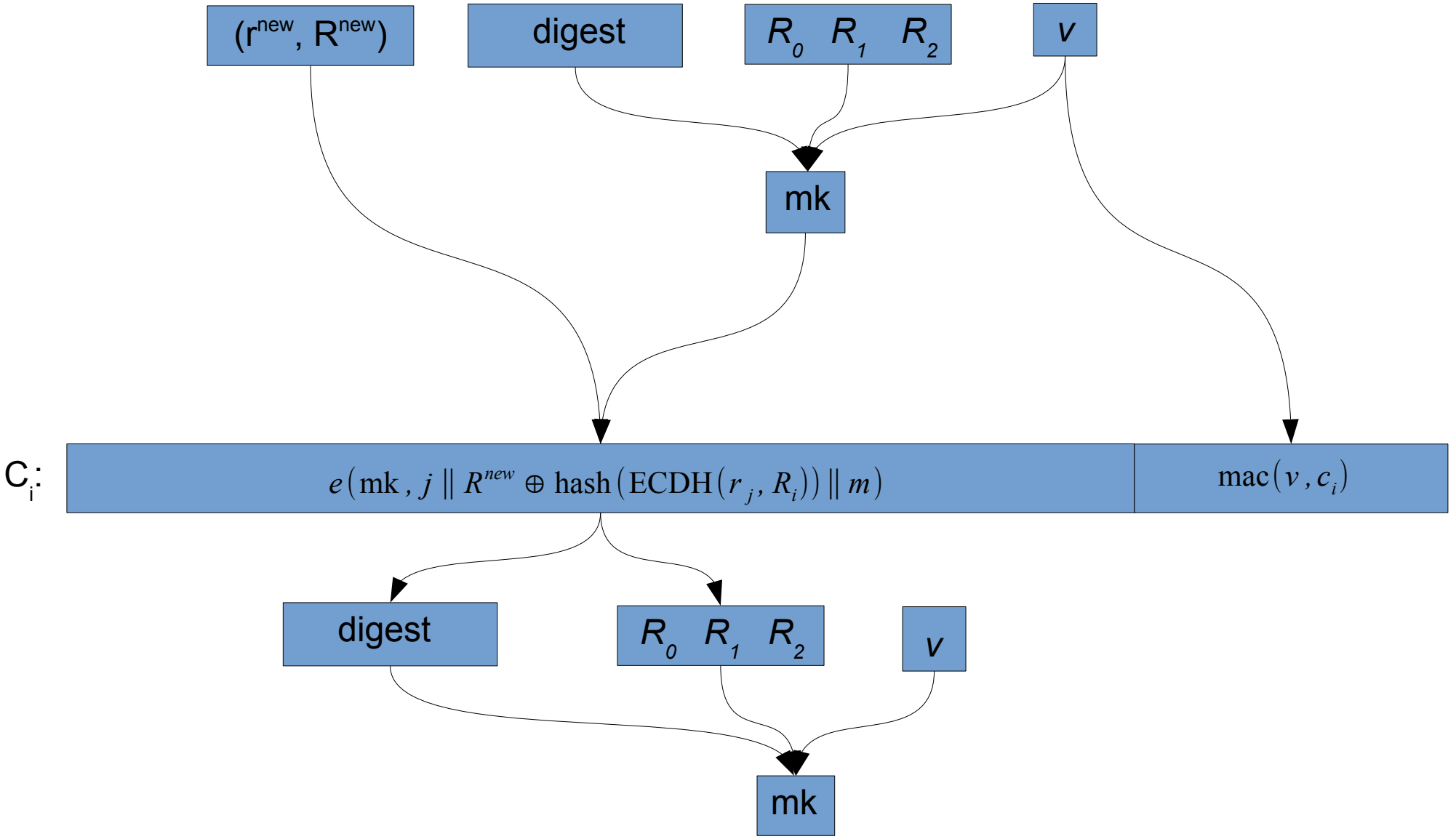
Routing server model

Router is not part of group and does not have encryption/decryption capabilities

Routing server leaks no more metadata than a fully-connected graph does to a GPA (transport metadata only)



SMMP – Message format (from P_j)



SMMP – Send

- P_j creates a message *m*.
- P_j generates a new ephemeral ratchet key: $\left(r_j^{new}, R_j^{new}\right)$
- P_j computes a new conversation digest:

$$\text{conv\_digest} = \text{hash}\left(m\right) \oplus \text{conv\_digest}$$

- For all $i \neq j$, P_j computes the set of *N-1* preliminary ciphertexts:

$$c_i' = e\left(\text{mk}, j \parallel R_j^{new} \oplus \text{hash}\left(\text{ECDH}\left(r_j, R_i\right)\right) \parallel m\right)$$

- P_j computes the set of *N-1* MACs: $c_{\text{mac}\,i} = \text{mac}\left(v, c_i'\right)$
- P_j forms the set of *N-1* ciphertexts: $c_i = c_i' \parallel c_{\text{mac}\,i}$
- P_j sends c_i to P_i for all all participants using a collision-avoidance algorithm appropriate for the transport.

SMMP – Send continued

- P_j updates his ratchet state:

$$(r_j, R_j) = (r_j^{new}, R_j^{new})$$

$$\mathrm{RK} = \mathrm{hash}\left(\mathrm{RK} \,\|\, \mathrm{ECDH}\left(v, \mathrm{hash}\left(\mathrm{conv\_digest} \,\|\, \|_i R_i\right)\right)\right)$$

$$\mathrm{mk} = \mathrm{KDF}\left(\mathrm{RK}, 0\mathrm{x}01\right)$$

Notation: $\|_i R_i = R_0 \,\|\, R_1 \,\|\, R_2 \,\|\, \ldots$

SMMP - Receive

- Upon receipt of a ciphertext *c*, P_j separates it into *c′* and c_{mac} parts.
- P_j tests if hash(*v, c′*) = c_{mac}. If they do not match, he raises a *Bad_HMAC* exception and goes no further.
- P_j obtains *q* || *R* || m = *d*(mk, *c′*). If this operation fails, he raises a *Message_Undecryptable* exception, sets the resync_required flag *True*, and passes *c′* and control to the message-received housekeeping routine.
- P_j computes a new conversation digest:

$$\text{conv\_digest} = \text{hash}(m) \oplus \text{conv\_digest}$$

and sets:

$$R_q = R \oplus \text{hash}(\text{ECDH}(r_j, R_q))$$

SMMP – Receive continued

- P_j updates his ratchet state:

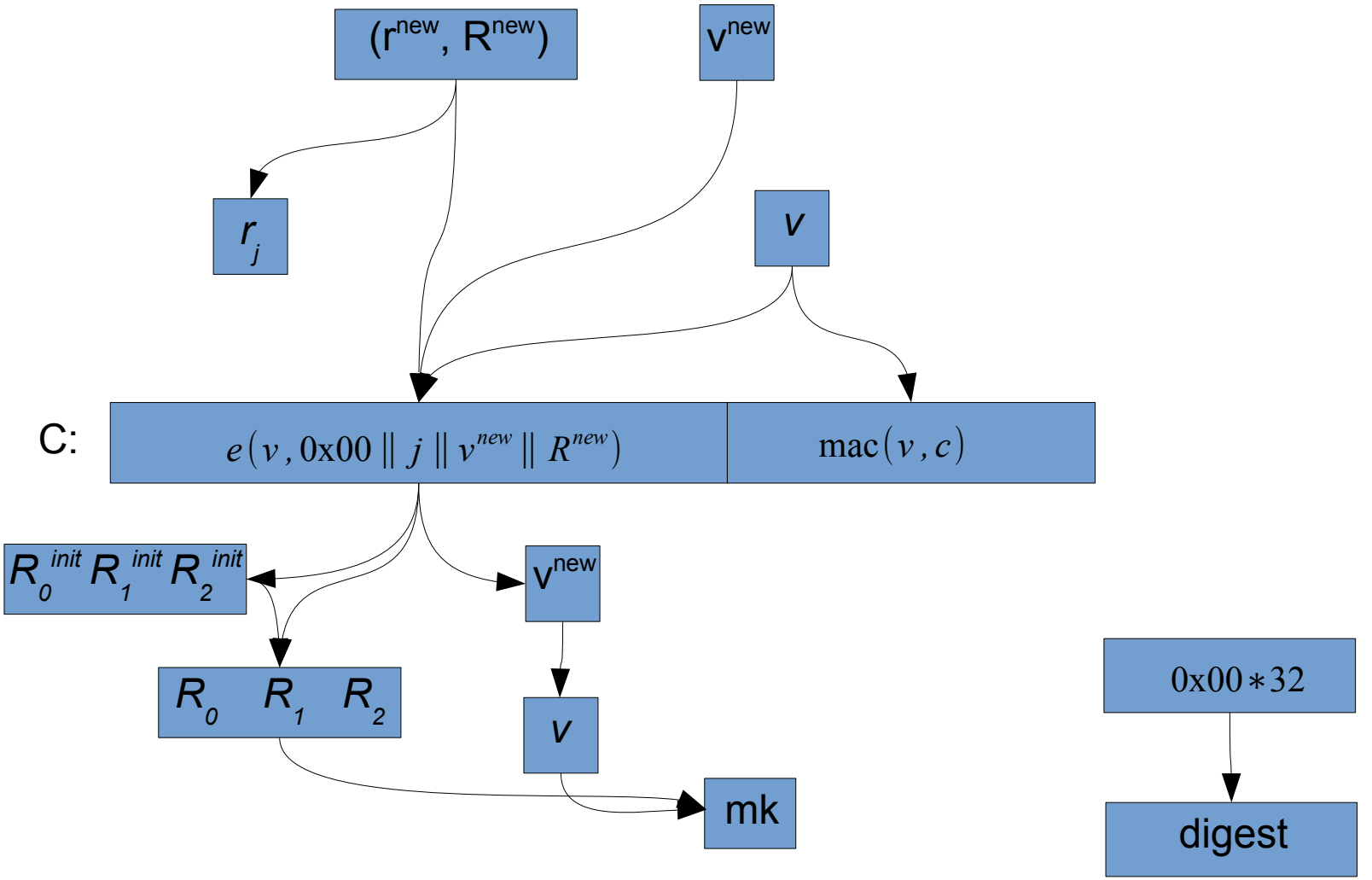
$$\mathrm{RK} = \mathrm{hash}\left(\mathrm{RK} \parallel \mathrm{ECDH}\left(v\,,\, \mathrm{hash}\left(\mathrm{conv\_digest} \parallel \|_i R_i\right)\right)\right)$$

$$\mathrm{mk} = \mathrm{KDF}\left(\mathrm{RK}\,,\, 0\mathrm{x}01\right)$$

SMMP - Housekeeping

- Resynchronization
  - Necessary due to dropped and broken messages (collisions, network failures, bad actors)
  - Send
  - Receive
- Add/remove participant
  - Re-run group key agreement for add/remove (moving to a different room)
  - Participant may drop themselves by simply not updating ratchet state
- IM within SMMP (whispering)
  - Bootstrap a private conversation using SMMP infrastructure

SMMP – Resync message format (from P_j)



SMMP – Resynchronization - Send

- P_j generates a new group private key v^{new}.
- P_j generates a new ratchet key pair (r^{new}, R^{new}).
- P_j computes preliminary ciphertext:

$$c' = e\left(v, 0\text{x}00 \parallel j \parallel v^{new} \parallel R^{new}\right)$$

- P_j computes mac: $c_{\text{mac}} = \text{mac}\left(v, c'\right)$
- P_j forms: $c = c' \parallel c_{\text{mac}}$
- P_j delays according to a transport-specific collision-avoidance algorithm.
- P_j tests resync_required. If *False*, a resynchronization message was received and no further action is necessary. If *True*, P_j broadcasts c to all participants P_i.

Resynchronization – Send continued

- Pj updates his ratchet state as follows:

$$v = \text{hash}\left(v \,\|\, v^{new}\right)$$

$$r_j = r^{new}$$

$$r_j^{init} = r^{new}$$

$$R_j^{init} = R^{new}$$

$$\left\{R_i\right\} = \left\{R_i^{init}\right\}$$

$$\text{RK} = \text{hash}\left(v \,\|\, \text{ECDH}\left(v\,,\, \|_i R_i\right)\right)$$

$$\text{mk} = \text{KDF}\left(\text{RK}\,,\, 0\text{x}01\right)$$

$$\text{conv\_digest} = 0\text{x}00 * 32$$

$$\text{resync\_required} = \mathit{False}$$

SMMP – Resynchronization - Receive

- P_j obtains *c'* from a failed Receive protocol
- P_j computes $m = q \,||\, v_{new} \,||\, R_q^{new} = d(v,c')$ and sets: $v = \text{hash}(v \,||\, v_{new})$

  $$R_q^{init} = R_q^{new} \text{ and } \{R_i\} = \{R_i^{init}\}$$

  If this decryption fails, P_j raises a *Message_Undecryptable* exception and goes no further
- P_j updates his ratchet state:

  $$\text{RK} = \text{hash}(v \,||\, \text{ECDH}(v, ||_i R_i)) \qquad \text{resync\_required} = \mathit{False}$$

  $$\text{mk} = \text{KDF}(\text{RK}, 0\text{x}01) \qquad \text{conv\_digest} = 0\text{x}00 * 32$$

SMMP – Security Analysis

- Security adversary
  - The computationally-bounded security adversary cannot control $\{R_i\}$ or v and has access only to the plaintext he was able to recover from one or more participants post-conversation. Therefore if the underlying symmetric encryption algorithm is secure against a known-plaintext attack, SMMP is secure. Further, due to inclusion of new (random) key material with each message, a key compromise at any stage of the conversation will not compromise keys for messages earlier or later in the conversation chain, resulting in PFS and PFuS.
- Consensus adversary
  - A group conversation digest is updated and passed with each message, therefore if the hash function used to compute the conversation digest has collision resistance a participant will not be able to reach consensus given a fake message.
- Privacy adversary
  - Group key agreement is based only on knowledge of a participant's public keys, therefore a fake transcript involving an honest participant can be created without his participation. This provides information-theoretic PD for both online and offline judges.

SMMP – Efficiency Analysis

- Only 3 rounds are required for authenticated GKA.
  - GKA is only required once – the ratchet advances with new key material from participants at each message.
- Each participant only manages the state of a single node in the network.
- GKA and message exchange are $O(N)$ in complexity.
- SMMP represents a significant improvement over improved-GOTR [Liu, *et.al.*], both for initial GKA, as well as during messaging.

Three may keep a secret,
if two of them are dead.

Benjamin Franklin,
Poor Richard's Almanack

Demonstration

- Ratchet visualization
  - Simulated conversation with eight participants
  - Observe changes in ratchet keyset and state
  - Observe resynchronization
- Chat client
  - Arbitrary number of participants
  - Uses routing server – server may run as TOR hidden service if desired to obfuscate transport metadata
  - Collision avoidance algorithm is TDMA
  - ncurses-based, written in Python