Improved Group Off-the-Record Messaging

Hong Liu
Department of Computing and Information Sciences
Kansas State University
hongl@ksu.edu

Eugene Y. Vasserman
Department of Computing and Information Sciences
Kansas State University
eyv@ksu.edu

Nicholas Hopper
Department of Computer Science and Engineering
University of Minnesota
hopper@cs.umn.edu

ABSTRACT

Off-the-Record Messaging (OTR) is an online analogy of face-to-face private chat – messages are confidential and authenticated at the time of the conversation, but cannot later be used to prove authorship. The original OTR protocol is limited to two parties, and is extended by multi-party OTR (mpOTR) to the group chat setting. In doing this, mpOTR unintentionally weakens the security properties provided by its two-party predecessor. We propose an improved group OTR (GOTR) protocol that provides unconditional repudiability, and show how to obtain data origin authentication given this level of repudiability.

GOTR resists network failure, colluding and independent malicious insiders, and provides efficient and flexible membership management. We analyze the security properties and performance of GOTR, and present measurement results of a proof-of-concept implementation of GOTR.

Categories and Subject Descriptors

K.4.1 [**Computers and Society**]: Public Policy Issues—*Privacy*; K.6.5 [**Computers and Society**]: Security and Protection—*Authentication*; H.4.3 [**Information Systems Applications**]: Communications Applications—*Computer conferencing, teleconferencing, and videoconferencing*; C.4 [**Performance of Systems**]: Fault tolerance

Keywords

Privacy; Repudiability; Group communication; Authentication; Robustness

1. INTRODUCTION

Off-the-Record Messaging (OTR) is a technology that protects the privacy of instant messaging users by mimicking crucial features of face-to-face conversations [2, 5, 14]: OTR users (let's call them Alice and Bob) communicate using an authenticated and confidential channel, but the content of the conversation is repudiable afterwards. Communication is bootstrapped using Alice's and Bob's long-term asymmetric keys, authenticating Alice to Bob and Bob to Alice, but the conversation itself is secured using symmetric cryptography, and thus repudiable in the sense that given a protocol transcript recorded by either Alice, Bob, or a 3rd party observer, no one, including Alice or Bob, can prove who authored a particular message in the transcript.

Unfortunately, it is nontrivial to extend two-party OTR to a group setting. Most notably, unlike pairwise OTR, group chat users need a way to know the identity of the sender amongst group members, namely the data origin authentication problem, but adding this feature may in turn break users' repudiability. Besides, membership management, colluding malicious parties, and network partitioning are problems specific to the group setting.

Goldberg et al. proposed a multi-party OTR protocol (mpOTR) which provides repudiation by using two-party deniable entity authentication, while keeping chat messages non-repudiable among group members during protocol execution [10]. A user Alice generates an ephemeral signature key pair, which is used only in the current session for Alice to sign her messages. To provide repudiation, users exchange ephemeral public signature keys in a deniable fashion, which is achieved by by using a two-party deniable authenticated key exchange (AKE) in the protocol setup. Deniable AKE convinces two parties of each other's identity during the protocol execution, but does not allow either party to convince anyone else that authentication was successful, and therefore provides repudiability.

However, mpOTR provides repudiability properties which are weaker than the two-party version. First, deniable authentication has many partially conflicting definitions which differ in subtle but important ways [1, 5, 6, 8, 9]. One of the more important distinctions is between *online and offline judges*. A judge is a third party who determines whether a given user has authenticated some message, including key exchange parameters. Online judges are allowed to interact with insider adversaries during the protocol execution, while offline judges can only interact afterward. This is conceptually similar to proving something in real time, versus after the fact using an historical transcript. Second, an mpOTR transcript *links all messages together*, so if a user, e.g. during an interrogation, shows her knowledge of any secret contained in the conversation, she cannot deny that she has participated in the protocol run, and therefore cannot repudiate the authorship of any messages she sent, even if the protocol ensures plausible deniability. Third, to forge an mpOTR transcript, a subgroup of participants must collude. While

WPES'13, November 4, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2485-4/13/11 ...$15.00.
http://dx.doi.org/10.1145/2517840.2517867.

not a problem if any single party is able to forge the transcript alone, in the multiparty setting it may be possible to show that two groups of mpOTR users act independently, or are unlikely to have colluded, severely hampering real-world plausible deniability.

We propose a group OTR (GOTR) protocol which enhances both real-world and protocol-level repudiability of group chats. GOTR achieves the same level of repudiability as that achieved using shared group keys (i.e. information-theoretic repudiability), while still preventing malicious insiders from forging a message in the name of an honest user. GOTR utilizes a special feature of some group key agreement (GKA) protocols to turn a GKA into a robust "hot-plug" GKA, and to further turn the adverse facts that the network may fail and users may be faulty into the security of the senders. Furthermore, the repudiability properties of GOTR do not rely on deniability of entity authentication algorithms. Most operations, such as key update, run symmetrically and simultaneously for each group member, and require a constant number of communication rounds. The protocol is also flexible in membership management and is resilient to membership churn. We instantiate our protocol using the Burmester-Desmedt (BD) GKA [3] and benchmark a proof-of-concept implementation of the protocol core as a plugin for Pidgin.[1]

2. GROUP OTR PROTOCOL

In this section we give descriptions of each subroutine of GOTR. We evaluate the efficiency and security properties of GOTR in the next section.

The GOTR protocol consists of the following subprotocols:

- `Setup`: In the setup protocol, a new user joins the group, establishes authenticated, private channels with every other user, and obtains a cryptographic digest of the conversation so far.
- `KeyUpdate`: Each user U_i runs a group key agreement protocol with all other users to obtain a symmetric *circle key* that every user knows, but only U_i will use to encrypt and authenticate messages. The information exchanged in this protocol will allow users to deduce the owner of U_i's circle key.
- `SendMsg`: When U_i has a message m to send, she uses her circle key to encrypt and authenticate the message, along with the chat digest, before sending the ciphertext to the group. After sending, U_i updates her digest.
- `RecvMsg`: When receiving a ciphertext c, U_j deduces the appropriate circle key from the presence of messages sent on private channels in `KeyUpdate`; the message is decrypted and integrity checked using this circle key. U_j provisionally updates his digest with the ciphertext c. The content of the message is not authenticated until the consistency check.
- `ConsCheck`: After receiving a ciphertext, each user exchanges digests with every user over private channels. If all members have the same digest, the message is considered to be authenticated. Since a failed consistency check could also be caused by network failure or an insider attack on chat consistency, and a sender could choose to include a forged message in her digest, plausible repudiability is maintained while allowing each user to detect forgeries using her circle key.

We note that no special procedure is needed to leave the group: when a user does not respond to `KeyUpdate` requests, she is assumed to have left the chat; the "hot-plug" nature of the `KeyUpdate` protocol allows automated recovery from such membership changes. If this situation results from a temporary network failure, the user can always re-join the group by running `Setup` again.

2.1 Primitives

We assume the existence of an anonymous point-to-point network between users. We also assume the existence of the following cryptographic primitives:

- $r \leftarrow Z_q$: generates a random value in Z_q.
- $SecSend(i, j, m)$, $m \leftarrow SecRecv(i, j)$: arbitrary two-party secure communication primitives that provide confidential and mutually authenticated message transmission between U_i and U_j over a point-to-point network. They can be implemented by using users' long-term asymmetric keys, mutually authenticated SSL/TLS, HMQV [13], or KEIA [8]. The algorithm does not have to be repudiable, deniable, or forward-secret.
- $\Sigma = (Gen, EncMac, Dec)$: a CCA-CMA secure [7] private-key message transmission scheme.
- $(k_1, k_2) \leftarrow KDF(\kappa)$: a Key Derivation Function which generates random keys (k_1, k_2) for encryption Enc and authentication Mac respectively.

2.2 Setup

The setup stage begins after a user U_i joins the chat. U_i first obtains the common parameters of the chat, including a unique session ID *sid*, membership set *pid*, and global cryptographic parameters. We assume that these parameters are obtained from another application protocol and are shared by all group members. U_i then performs mutual entity authentication with every other user and establishes end-to-end secure communication channels.

2.3 Key Update

In key update, U_i refreshes the keys which will be used to protect her messages in the communication stage, by performing a "hotplug" GKA. Key update is invoked asynchronously by any user. The GKA is hotplug, in the sense that a party participating in the GKA can dynamically add or remove parties from the GKA without the need to recommunicate. For example, we can turn the Burmester-Desmedt GKA (BD GKA) [3] into a hotplug GKA.

To avoid ambiguity, we call each entity that participates in the BD GKA a "node". The BD GKA assumes a broadcast channel (a requirement we later relax). Nodes $\mathcal{D} = \{D_1, \ldots, D_n\}$ negotiate a key κ in three steps:

1. $\forall D_l \in \mathcal{D}$, D_l generates a secret key $r_l \in Z_q$, computes and broadcasts the public key:

$$z_l = g^{r_l} \bmod p, \tag{1}$$

where p is a prime, and the subgroup $\langle g \rangle$ in Z_p^* is of prime order q and satisfies the decisional Diffie-Hellman assumption.

[1] Source code available from `http://www.cis.ksu.edu/~hongl/gotr/`

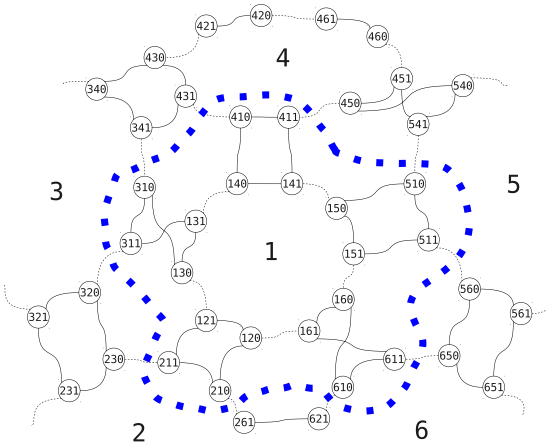

Figure 1: Key graph of the "hotplug" GKA: virtual nodes within the bold dashed line compose U_1's circle key. Each virtual node is labeled $(ijl), l \in \{0, 1\}$, which denotes the virtual node D_{ijl} that is generated by U_i to talk with U_j.

2. Every D_l computes and broadcasts

$$X_l = (z_{l+1}/z_{l-1})^{r_l} \bmod p, \quad (2)$$

where the indices are taken in a cycle.

3. Every D_l obtains the same key

$$\kappa = \kappa_l = (z_{l-1})^{nr_l} \cdot X_l^{n-1} \cdot X_{l+1}^{n-2} \cdots X_{l-2} \bmod p. \quad (3)$$

In the following text we will sometimes refer to the public keys computed from equation 1 as "z values", and the values computed from equation 2 as "X values". The GKA is proved secure (κ is indistinguishable from a random session key) against a computationally-bounded eavesdropper with transcripts of polynomially-many executions of the GKA [4, 11].

We observe that BD has the following special properties:

- It is symmetric, in the sense that every node performs the same operations;
- The group key is derived from the data that every node provides;
- Contributory data $\{z_l, X_l\}$ provided by any node D_l only depends on a constant number of nodes (i.e. two).

We rely on these properties to obtain a hotplug GKA, so that the GKA can be performed over a point-to-point network. We ask every user U_i to act as two "virtual nodes" (D_{ij0} and D_{ij1}) when performing contributory data exchange with user U_j, i.e. D_{ij0} and D_{ij1} will talk to D_{ji0} and D_{ji1} as shown in Figure 1. We will use the subscript $ijl, l \in \{0, 1\}$ to denote a value associated with the virtual node D_{ijl}. We will denote (a_{ij0}, a_{ij1}) as a_{ij} and $\{a_{ij} | \forall U_j \in pid \setminus \{U_i\}\}$ as a_i for short.

First U_i generates r_{ij} and sends $z_{ij} = (g^{r_{ij0}}, g^{r_{ij1}})$ to U_j. In response, U_j sends z_{ji} to U_i, and U_i stores them in $y_{ij} = z_{ji}$. To obtain a group key, **the X values must connect $\mathcal{D}$ into a circle**. X_l relies on its own secret key r_l, and its two neighboring nodes' public keys.

To allow U_j to obtain U_j's BD keys, U_i computes the X values for the triples $(D_{ji0}, D_{ij0}, D_{ij1})$ and $(D_{ij0}, D_{ij1}, D_{ji1})$, respectively:

$$R_{ij0} = (z_{ij1}/y_{ij0})^{r_{ij0}} \bmod p$$

$$R_{ij1} = (y_{ij1}/z_{ij0})^{r_{ij1}} \bmod p.$$

U_i then sends R_{ij} to U_j. U_j does the same for U_i, and U_i stores U_j's R values in $V_{ij} = R_{ji}$.

At this point, U_i and U_j have exchanged all the contributory data needed to compute the BD key for a group of four virtual nodes. We call this four-node group a "flake". In Figure 1, the flake that is formed by user U_1 and U_4, denoted as $flake_{14}$, is composed of the virtual nodes ($D_{140} \rightarrow D_{410} \rightarrow D_{411} \rightarrow D_{141} \rightarrow D_{140}$). Following the BD algorithm in equation (3), the group key for $flake_{ij}$, called a "flake key", can be computed as follows:

$$\kappa_{ij} = z_{ij1}^{4 \cdot r_{ij0}} \cdot (\frac{1}{R_{ij0}})^3 \cdot V_{ij0}^2 \cdot V_{ij1} \bmod p$$

$$= \kappa_{ji} = z_{ji1}^{4 \cdot r_{ji0}} \cdot (\frac{1}{R_{ji0}})^3 \cdot V_{ji0}^2 \cdot V_{ji1} \bmod p$$

Then U_i, U_j verify that they compute the same key $\kappa_{ij} = \kappa_{ji}$ with (z, y, R, V) bound with their identities and the session, as shown in Algorithm 1. This also ensures that both parties transmitted the correct $\{z, X\}$ pairs. Because the BD GKA is secure in the presence of eavesdroppers with a history transcript, if U_i replays the $\{z, X\}$ pairs from another user, or provides incorrect data to U_j, she cannot obtain a shared key with U_j. As an optimization, instead of using a digital signature, the flake key κ_{ij} can then be used to secure the end-to-end channel between U_i and U_j.

Now U_i has obtained $(N-1)$ disjoint flakes. She proceeds to re-compute the X values for D_{ij0} and D_{ij1} to connect the flakes. To do so, U_i computes $W_{ij}, \forall U_j \in pid \setminus \{U_i\}$, which are the X values of the triples $(D_{i(j-1)1}, D_{ij0}, D_{ji0})$ and $(D_{ji1}, D_{ij1}, D_{i(j+1)0})$, using r_i, z_i, and y_i:

$$W_{ij0} = (y_{ij0}/z_{i(j-1)1})^{r_{ij0}} \bmod p$$

$$W_{ij1} = (z_{i(j+1)0}/y_{ij1})^{r_{ij1}} \bmod p.$$

Note that U_i computes W_i by herself without communicating with other users. The indices and the order do not matter (as long as the computation follows one direction; we can change the direction of an X value by inverting it).

Using any element in r_i and the corresponding element in z_i, together with W_i and V_i, U_i is able to compute her BD key K_i, which is the BD group key of $4(N-1)$ virtual nodes. For the group of users $\{U_1, \ldots, U_6\}$ in Figure 1, U_1 obtains a circle of virtual nodes ($D_{140} \rightarrow D_{410} \rightarrow D_{411} \rightarrow D_{141} \rightarrow D_{150} \rightarrow D_{510} \rightarrow \ldots \rightarrow D_{130} \rightarrow D_{310} \rightarrow D_{311} \rightarrow D_{131} \rightarrow D_{140}$). Applying equation (3), U_1 can use z_{131}, r_{140}, W_{140}, V_{140}, V_{141}, W_{141}, W_{150}, V_{150}, ..., W_{130}, V_{130}, V_{131}, to compute the BD key of $4 \cdot (6-1)$ virtual nodes:

$$K_1 = z_{131}^{4 \cdot 5 \cdot r_{140}} \cdot W_{140}^{4 \cdot 5 - 1} \cdot V_{140}^{4 \cdot 5 - 2} \cdot V_{141}^{4 \cdot 5 - 3} \cdot W_{141}^{4 \cdot 5 - 4} \cdot W_{150}^{4 \cdot 5 - 5} \cdots V_{131} \bmod p.$$

Alternatively, U_1 can use z_{161}, r_{120}, W_{120}, ..., V_{161}, or other (z, r, W, V) combinations to compute K_1.

The above operation is the same for everybody. In all, N different group keys will be generated, one for each user. In order to differentiate them from the group key in its conventional notion, we call them "circle keys". Up to now, every user obtains a unique circle key that is negotiated by the entire group, but does not know other users' circle keys since W_i is essential to connect the circle, but is computed by U_i alone. To obtain unconditional repudiability, every user should know (z_i, y_i, W_i, V_i) (called $zyWV$ for short). We therefore ask every user to "broadcast" $zyWV$. Upon

```
Data: $\forall U_i \in pid, sid, g, p, q$
Result: $r_i, z_i, y_i, R_i, V_i, \kappa_i, W_i, K_i$
Keyupdate($i$, $g$, $p$, $q$)
begin
    for $\forall U_j \in pid \setminus \{U_i\}$ do
            /* this is performed concurrently */
        $r'_{ij0} \leftarrow Z_q$, $r'_{ij1} \leftarrow Z_q$
        SecSend($i$, $j$, $sid|z'_{ij}$)
        $sid|y'_{ij} \leftarrow$ SecRecv($i$,$j$)
        SecSend($i$, $j$, $sid|R'_{ij}$)
        $sid|V'_{ij} \leftarrow$ SecRecv($i$,$j$)
        compute $\kappa'_{ij}$
        $(k_1, k_2) \leftarrow KDF(\kappa'_{ij})$
        SecSend($i$, $j$,
        $sid|Mac_{k_2}(sid|U_i|U_j|z'_{ij}|y'_{ij}|R'_{ij}|V'_{ij})$)
        if $sid|Mac_{k_2}(sid|U_j|U_i|y'_{ij}|z'_{ij}|V'_{ij}|R'_{ij}) =$
        SecRecv($i$,$j$) then
            update $r_{ij}$, $z_{ij}$, $y_{ij}$, $R_{ij}$, $V_{ij}$
    for $\forall U_j \in pid \setminus \{U_i\}$ do
        compute $W_{ij}$
    if message to send then
        for $\forall U_j \in pid \setminus \{U_i\}$ do
            SecSend($i$,$j$, $sid|z_i|y_i|W_i|V_i$)
        compute $K_i$
    else
        for $\forall U_j \in pid \setminus \{U_i\}$ do
            randomize elements $z'_{il}$, $y'_{il}$, $l \neq j$
            re-compute $W'_i$, $V'_i$
            SecSend($i$,$j$, $sid|z'_i|y'_i|W'_i|V'_i$)
```

Algorithm 1: `Keyupdate`: refresh the circle key

receiving U_i's $zyWV$, any U_j can compute U_i's BD key K_i, using the private part $r_{jil}, l \in \{0,1\}$ of $z_{jil} \in y_i$.

If U_i fails to talk to any U_j in above steps, she will not obtain the data of D_{ji0} and D_{ji1}, but since she controls $2(N-1)$ virtual nodes within the circle, she can arrange the circle as she wishes by changing the neighbors of her virtual nodes. This allows U_i to do the "hotplugging" trick: the $\{z, X\}$ pairs that any U_j provided can be safely removed or replaced, and everybody except U_j is still able to compute the same key with U_i. Specifically, to remove U_j from the circle, U_i removes $\{D_{ij0}, D_{ji0}, D_{ji1}, D_{ij1}\}$ altogether from the circle and computes the X values for $D_{i(j-1)1}$ and $D_{i(j+1)0}$ using the triples ($D_{(j-1)i1}$, $D_{i(j-1)1}$, $D_{i(j+1)0}$) and ($D_{i(j-1)1}$, $D_{i(j+1)0}$, $D_{(j+1)i0}$) respectively. A valid but different circle key will be generated without affecting other users. In a similar way, U_i can replaces U_j's data, and can also insert an arbitrary number of virtual nodes into her circle to generate a different key.

Hotplugging provides robustness against insider attacks and network failure. In addition, since every user acts as different virtual nodes when talking to different users, data origin authentication is possible. This feature also enables an optimization to enhance robustness against inside forgers of limited power. If U_i modifies $zyWV$ by removing the neighbors of D_{ij0} and D_{ij1} and re-computing the corresponding W values, then U_j will not be able to compute U_i's circle key. We therefore ask U_i to compute and send the real $zyWV$ only if she is going to send a message before her next key update. Otherwise she removes or replaces the $\{z, X\}$ values from some $U_j \neq U_k$, re-computes the corresponding W values, and sends the modified copy of $zyWV$ to U_k. Because the circle key is hotplug, the modified $zyWV$ is still legitimate. U_k cannot tell if the $\{z, X\}$ pairs are real or not, but if U_k sends U_j the modified $zyWV$ with a message encrypted with the BD key derived from them, U_j will not be able to compute a key and read the message. This prevents inside forgers who act individually and do not control message delivery, like average end-users, from abusing U_i's circle key. Note that ultimately we prevent forgery by any inside forgers through conversation consistency checks. This optimization enhances robustness by defending less powerful forgers from easily disrupting the protocol execution, but is not required to achieve the desired security properties.

In practice, when a human user is typing, the GOTR client knows that she is planning to send a message, and will invoke a key update if the key is old enough. Message sending may be slightly delayed in order to use the new key.

2.4 Communication: `SendMsg`, `RecvMsg`, `ConsCheck`

U_i uses the keys derived from her circle key to encrypt and authenticate messages. All the $\{z, X\}$ pairs needed to compute the circle key (i.e. $zyWV$) piggyback on the message, together with the digest (as a consistency check) of the chat log up to but not including this message. We emphasize that the digest of the chat log is performed over the chat message ciphertext (without decryption and with the $\{z, X\}$ pairs) from U_i's view of the chat. In particular, the chat log does not include data transmitted in other operations such as key update. The whole message is then sent directly to the underlying (anonymous) messaging service, or to each user one by one over an anonymous network.

Specifically, U_i's message is in the following format:

$$sid \mid Op \mid clen \mid z_1, X_1, \ldots, z_{clen}, X_{clen} \mid$$
$$Enc_{k_1}(m, padding, digest) \mid Mac_{k_2}(sid..Enc)$$

where sid is the session id, Op is a marker indicating that the message is a chat message, and $clen$ is the number of $\{z, X\}$ pairs that follow. It is also the number of exponentiations needed to compute the key. K is a BD key computed from the $\{z, X\}$ pairs, together with the private part of one element in $\{z_i | i = 1, \ldots, clen\}$, which is exactly the honest sender U_i's $zyWV$, and K is her circle key. k_1, k_2 are derived by feeding a Key Derivation Function with K, and are used for encryption Enc and authentication Mac respectively. $digest$ is the sender's view of the chat log digest. The Mac tag is computed over the entire message, starting from the sid field through the end of the ciphertext.

This design is multipurpose. First, piggybacking $\{z, X\}$ pairs on the message does not break the secrecy of the key. Second, it guarantees that all receivers who contributed to U_i's circle key can decrypt the message and verify its integrity, regardless of whether they have received the previously "broadcast" data or not. Third, receivers rely on the $\{z, X\}$ pairs to determine the message origin (explained below). Fourth, the piggybacking $\{z, X\}$ pairs provides some robustness against DoS attacks, without the need to verify the Mac. Fifth, the digest provides a convenient way to perform a consistency check on the chat views without revealing any chat content or sender identities. Last but not least, computing the digest over the ciphertext instead of the plaintext provides robustness to failure and resilience to attacks on setup and key update.

Upon receiving a message, U_j matches $\{z_i, X_i | i = 1, \ldots, clen\}$ to his z_j and R_j, and records U_l if $z_{jl}, R_{jl} \in$

$\{z_i, X_i | i = 1, \ldots, clen\}$. If he finds at least one matching (z_{jl}, R_{jl}), he is able to compute the key and regards the matching U_l('s) as the message sender(s). (Because insiders may collude and share secrets, it is possible to identify more than one potential sender.) He then checks the message integrity and decrypts m and the digest. U_j will confirm the sender identity at the next consistency check with U_l over the pairwise secure channel. If m is not from U_l, U_l creates a consistency check failure to indicate that m is forged.

3. EVALUATION OF GOTR

3.1 Security Analysis

The high-level security goal of GOTR is to realize a "private chatroom" over a public network. This includes several conventional goals against outsiders such as confidentiality, integrity, robustness and loose synchrony; for space concerns we do not further consider these goals here. Instead we focus on the ways in which GOTR differs from prior work on OTR in a group setting: repudiability and authentication. Throughout our analysis, we consider a group of N parties that can communicate pairwise over an insecure network. We assume a method for pairwise authenticated key establishment, access to an anonymizing network, and allow concurrent protocol executions, each identified by a common (public) session identifier. We distinguish between a "chat transcript" that lists for each party in a session, the input messages delivered to the `sendMsg` subroutine and times; and a "protocol transcript" that includes all messages exchanged by the GOTR protocol.

Offline Repudiability. Briefly, we define strong offline repudiability in terms of an *adversary*, who controls $N-1$ parties and produces a pair of chat transcripts τ, τ' that differ in a single message; a single honest party that faithfully executes the GOTR protocol with the adversarial parties given a chat transcript τ; a *simulator* that takes as input a pair of chat transcripts τ, τ', a protocol transcript T corresponding to τ, and the private inputs s of the adversary, and produces a protocol transcript T' corresponding to τ'; and a *judge* that takes as input a chat transcript τ^*, protocol transcript T^*, and private inputs s of the adversary, and outputs a single-bit guess. A group messaging protocol has strong offline repudiability if for every adversary and judge, there exists a simulator such that the judge cannot distinguish between the case that (T^*, τ^*, s) result from the interaction of the adversary and the honest party on transcript τ and the case that (T^*, τ^*, s) result from the output of the simulator on input (T', τ', τ, s). Informally, the transcript may bind users to the conversation, but every single message in the conversation is repudiable after the fact.

We sketch the simulator S needed for our proof that GOTR provides strong offline repudiability. The simulator need not change anything in the transcript T up to the point at which τ and τ' differ. In the round in which they are different, S uses the private inputs of the adversarial parties along with the circle key broadcast of the honest user to derive the circle key and produce an appropriate encryption of the message in τ'. For all following rounds, S uses the symmetric keys shared by the adversarial parties with the honest user to simulate the honest user's inputs following τ' and runs the adversary code to produce the adversarial portion of the transcript T'. Note that since T and T' use the same secret keys, the output T' is identically distributed to an interaction between the adversary and the honest user with transcript τ'.

Online repudiability. We define online repudiability in terms of a *judge* that controls $N-2$ parties, honest parties H_1 and H_2 that faithfully execute the GOTR protocol given an adversarially chosen chat transcript τ, and an *environment* that controls two players, including the loss of network inputs to and from those players, and takes a chat transcript τ and adversarially chosen round r as input. We say that a protocol has online repudiability if for every judge there exists an environment so that the judge cannot distinguish between interacting with H_1 and H_2 on input τ and the environment on input τ; and the environment never produces a message from H_1 in round r. Informally, for every message sent by H_1 there must exist a *plausible alternative scenario* in which H_1 does not send the message, either via collusion with H_2 or as a result of network conditions. Note that this possibilistic notion of security against a stronger adversary is incomparable to offline repudiability: a protocol may be secure against either notion, both, or neither.

Informally, the environment Z required to show that GOTR has online repudiability works as follows. In rounds prior to r, Z executes H_1 and H_2 faithfully according to the transcript τ. At round r, Z uses the circle key computed by H_1 to send protocol messages via H_2 that decrypt to H_1's message in chat transcript τ. In subsequent rounds, Z simulates a complete outbound network failure for H_1, and uses the pairwise symmetric keys negotiated by H_1 with all judge-controlled parties to send key update messages and protocol messages via H_2, according to the chat transcript τ. Since Z knows all secrets associated to both parties, protocol messages sent to the judge will continue to have the same probability distribution as if H_1 and H_2 interact honestly. We note that this proof can be extended to the case that H_1 does not faithfully execute the protocol, for example by causing an unexpected consistency check failure or leaving a chat before the end of the transcript τ; these actions may be plausibly explained by network failures caused by Z.

Authentication. We define authenticated group messaging in terms of an adversary that controls $N-2$ parties and two honest parties H_1 and H_2. The adversary may run any number of concurrent sessions with H_1 and H_2, and request H_1 and H_2 to send any chat messages in those sessions, before sending a chat message m^* to H_1 in a target session s^*. The adversary wins if H_1 accepts m^* with source H_2, and m^* was never sent by H_2 in s^*. A group messaging protocol is authenticated if no efficient adversary can win this game.

To see that GOTR is authenticated, we consider the conditions required to accept m^* from H_2: (i) the circle key applied to m^* must include a z value sent by H_1 to H_2; (ii) the protocol message Mac must verify; and (iii) H_1 must receive a consistency check message from H_2 that includes m^* in the digest. We note that the security of the BD GKA prevents an adversary from deriving the circle key if a previous z value sent by H_1 is replayed in a different BD GKA message. Furthermore, the security of the Mac prevents the adversary from constructing a correct tag from an unknown key. Finally, if a message is encrypted using H_2's circle key and H_2 did not send the message, honest H_2 will never include m^* in the digest of a consistency check; forgery of such a message is impossible due to the security of the Mac used in pairwise communications and the security of the pairwise authenticated key agreement scheme.

3.2 Complexity

Leveraging the BD GKA, our scheme requires four rounds of contributory data exchange during a key update, plus two more rounds to verify the data using the flake key. The number of messages to exchange the contributory data for each user is $O(N)$. The message transmitted in the communication stage is of size $O(N)$ since all the contributory data is piggybacked with the message. To add a member to the group chat, each user only needs to add one flake into her circle, i.e. 3 more messages to send and receive per key update. To remove a member, each user removes one flake, i.e. no messages are transmitted. If the group only has two users, the flake is the circle, and our scheme operates in a way very similar to two-party OTR.

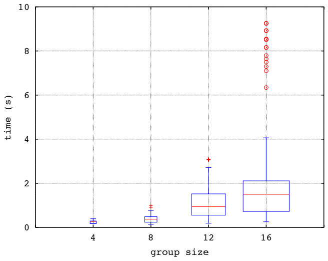

Figure 2: (a) End-to-end delay. Boxes indicate 1st and third quartiles, bands inside the box indicate medians, crosses and circles indicate outliers.

To measure the performance of our protocol in a more realistic environment, we implement GOTR as a Pidgin plugin, and run chats of 4, 8, 12, and 16 XMPP clients using a public server jabber.org. We use the 2048-bit MODP group as specified in RFC 3526 [12] to measure the execution time of the plugin on Intel Core i5-2500K computers running Linux. Measurement shows that the execution time is mostly due to the transmission delay from one Jabber client to another. The boxplots with whiskers with maximum 1.5 IQR of the data are shown in Figure 2.

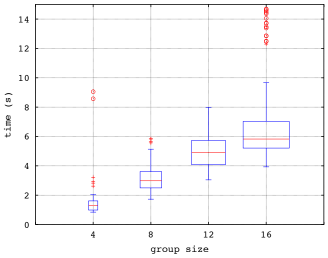

Figure 2: (b) Total execution time, single key update.

4. CONCLUSION

We proposed a group OTR protocol GOTR that provides unconditional repudiability, in the sense that we allow users to repudiate the authorship of session messages in the presence of a computationally unbounded online judge, with the assistance of a subset of insiders, while forgery by malicious insiders can still be detected. The repudiability of our scheme does not rely on the deniability of the entity authentication algorithm, and therefore circumvents the problems in real-world plausible deniability. GOTR utilizes a feature of certain GKAs, such as Burmester-Desmedt, to turn it into a robust "hotplug" GKA, and to further turn the adverse facts that the network may fail and users may be faulty into the security of the senders. We have implemented the core protocol as a Pidgin plugin.

Acknowledgements.

We thank Ian Goldberg, Yongdae Kim, and several anonymous referees for their helpful comments and discussions about this work. This research was partially supported by NSF grant 0917154.

References

[1] J.-M. Bohli and R. Steinwandt. Deniable group key agreement. In *Progress in Cryptology – VIETCRYPT 2006*, volume 4341 of *Lecture Notes in Computer Science*. 2006.

[2] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the ACM workshop on Privacy in the electronic society*, WPES '04, 2004.

[3] M. Burmester and Y. Desmedt. A secure and efficient conference key distribution system. In *Advances in Cryptology – EUROCRYPT '94*, volume 950 of *Lecture Notes in Computer Science*. 1995.

[4] M. Burmester and Y. Desmedt. A secure and scalable group key exchange system. *Information Processing Letters*, 94(3), 2005.

[5] M. Di Raimondo, R. Gennaro, and H. Krawczyk. Secure off-the-record messaging. In *Proceedings of the ACM workshop on Privacy in the electronic society*, WPES '05, 2005.

[6] M. Di Raimondo, R. Gennaro, and H. Krawczyk. Deniable authentication and key exchange. In *Proceedings of the ACM conference on Computer and communications security*, CCS '06, 2006.

[7] T. Diament, H. K. Lee, A. D. Keromytis, and M. Yung. The dual receiver cryptosystem and its applications. In *Proceedings of the ACM conference on Computer and communications security*, CCS '04, 2004.

[8] Y. Dodis, J. Katz, A. Smith, and S. Walfish. Composability and on-line deniability of authentication. In *Proceedings of the Theory of Cryptography Conference on Theory of Cryptography*, TCC '09, 2009.

[9] C. Dwork, M. Naor, and A. Sahai. Concurrent zero-knowledge. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, 1998.

[10] I. Goldberg, B. Ustaoğlu, M. D. Van Gundy, and H. Chen. Multi-party off-the-record messaging. In *Proceedings of the ACM conference on Computer and communications security*, CCS '09, 2009.

[11] J. Katz and M. Yung. Scalable protocols for authenticated group key exchange. *J. Cryptol.*, 20(1), 2007.

[12] T. Kivinen and M. Kojo. More modular exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). *RFC 3526*, 2003.

[13] H. Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*. 2005.

[14] M. Mannan and P. van Oorschot. A protocol for secure public instant messaging. In *Financial Cryptography and Data Security*, volume 4107 of *Lecture Notes in Computer Science*. 2006.