ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

Υλοποίηση ενός Off-the-Record Πρωτοκόλλου συνομιλίας πολλαπλών χρηστών

Συγγραφείς:
Κωνσταντίνος Ι. ΑΝΔΡΙΚΟΠΟΥΛΟΣ
Δημήτριος Θ. ΚΟΛΟΤΟΥΡΟΣ

Επιβλέπων:
Αριστείδης ΠΑΓΟΥΡΤΖΗΣ
Αν. Καθηγητής Ε.Μ.Π.

Διπλωματική εργασία

στη

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών

Αθήνα, Οκτώβριος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

Υλοποίηση ενός Off-the-Record Πρωτοκόλλου συνομιλίας πολλαπλών χρηστών

Συγγραφείς:
Κωνσταντίνος Ι. ΑΝΔΡΙΚΟΠΟΥΛΟΣ
Δημήτριος Θ. ΚΟΛΟΤΟΥΡΟΣ

Επιβλέπων:
Αριστείδης ΠΑΓΟΥΡΤΖΗΣ
Αν. Καθηγητής Ε.Μ.Π.

Διπλωματική εργασία

στη

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή:

....	....	....
Αριστείδης Παγουρτζής Αν. Καθηγητής Ε.Μ.Π.	Άγγελος Κιαγιάς Αν. Καθηγητής Ε.Κ.Π.Α.	Παναγιώτης Τσανάκας Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2016

....
Ανδρικόπουλος Ι. Κωνσταντίνος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών

....
Κολοτούρος Θ. Δημήτριος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών

Περίληψη

Σε έναν κόσμο όπου η ανάγκη για εύκολη και άμεση επικοινωνία πρέπει να ξεπεράσει την απειλή της συνεχούς παρακολούθησης, η κρυπτογράφηση από άκρο σε άκρο έχει καταστεί ανάγκη. Παρότι διάφορα πρωτόκολλα επιτρέπουν κρυπτογράφηση από άκρο σε άκρο για δύο άτομα, υπάρχουν λίγες μέθοδοι για την περίπτωση πολλαπλών ατόμων, ειδικά για εφαρμογές προσωπικού υπολογιστή. Σε αυτό το έργο περιγράφουμε την πρώτη υλοποίηση του mpOTR, του Multiparty OTR πρωτοκόλλου. Το πρωτόκολλο αυτό πετυχαίνει εμπιστευτικότητα, αυθεντικοποίηση, μυστικότητα προς-τα-εμπρός, και βασική ομοφωνία. Η υπάρχουσα θεωρητική περιγραφή του mpOTR μεταχειρίζεται κάποια υπο-πρωτόκολλα ως μαύρα κουτιά και δεν τα περιγράφει λεπτομερώς. Η δική μας συνεισφορά είναι η πλήρης και λεπτομερής περιγραφή του mpOTR πρωτοκόλλου, καθώς και η πρώτη του υλοποίηση. Η υλοποίηση αυτή είναι μια ανοικτού κώδικα, επιπέδου παραγωγής επέκταση της προϋπάρχουσας libotr βιβλιοθήκης, συνοδευόμενη και από ένα πρόσθετο Pidgin, γραμμένα και τα δύο σε C.

Λέξεις Κλειδιά: Ιδιωτικότητα, Ομαδικές Συνομιλίες, Αυθεντικοποίηση, Κρυπτογράφηση, Συνέπεια Συνομιλίας, Άκρο σε Άκρο, Παρακολούθηση

Abstract

In a world where the need for easy instant communication must overcome the threat of constant surveillance, end-to-end encryption has become a necessity. While some protocols enable end-to-end encryption for two parties, there are limited solutions for multiparty end-to-end encryption, particularly for desktop applications. In this paper we describe the first implementation of mpOTR, the multiparty OTR protocol. mpOTR achieves confidentiality, authenticity, forward secrecy, deniability, and basic consensus. The existing theoretical introduction of mpOTR treats the underlying subprotocols as black boxes and does not describe them in detail. Our contributions are the complete description of the mpOTR protocol including every subprotocol detail and the first implementation of mpOTR. Our implementation is a production-grade open source extension of the existing libotr library accompanied by a Pidgin plugin written in C.

Keywords: Privacy, Group Messaging, Authentication, Encrytpion, Trancript Consistency, End to End, Surveillance

Acknowledgements

We would like to thank our supervisor Professor Aris Pagourtzis, for giving us the opportunity to work on this project, and teaching us cryptography. We must also thank professor Angelos Kiayias for his recommendations which were always to the point.

Additionaly, this year-long journey wouldn't have met its end without the help of Dionysis Zindros, who guided us and sacrificed a great portion of his time to make sure that our work is finished.

Petros Angelatos, Kostis Karantias, and Eva Sarafianou have our gratitude for using our software during beta testing.

The Authors

My co-author, Dimitris, spent a lot of sleepless nights developing the code and solving problems with me or even alone when I couldn't join him. For this he deserves my gratitude as a person and respect as a software engineer.

While the following people were not directly involved in our work, I would never be able to complete this project without their help, teachings and support.

Nikos Papaspyrou, Dimitris Fotakis, and Vangelis Koukis played a major role in forming my mathematical and programming background. Without them my academic life would be extremely different.

I also can not leave out the Flute protocol's author, George Kadianakis. My engagement in Flute's development process helped me form a better understanding of the theoretical and practical aspects of secure instant messaging.

I would also like to thank my family, and my friends: Alexis L. and Alexis T., Angeliki, Vasilis K. and Vasilis H., Eleni, Ilias, Korina, Marilena, Orestis, and Panayiotis. Without them this thesis would have been completed much sooner.

Finally I must thank the members of the Free Software society of National Technical University of Athens for all the technical and philosophical discussion we had about software development.

Andrikopoulos Konstantinos

I would like to thank my co-author, Kostis, for the initial idea for the project, his advices on cryptography theory and the excellent collaboration.

Also, I owe a debt of gratitude to my parents for their full support all over the years.

Finally, I would like to thank my friends Pavlos, Lefteris and Panos who contributed to my emotional well-being throughout the completion of the thesis at hand.

Kolotouros Dimitrios

Contents

List of Algorithms

Listings

List of Figures

List of Abbreviations

ADT	**A**bstract **D**ata **T**ype
AES	**A**dvanced **E**ncryption **S**tandard
API	**A**pplication **P**rogramming **I**nterface
denAKE	**den**iable **A**uthenticated **K**ey **E**xchange
DSA	**D**igital **S**ignature **A**lgorithm
DSKE	**D**eniable **S**ignature **K**ey **E**xchange
ECB	**E**lectronic **C**ode **B**ook
EdDSA	**Ed**wards-curve **D**igital **S**ignature **A**lgorithm
GKA	**G**roup **K**ey **A**greement
IM	**I**nstant **M**essaging
IRC	**I**nternet **R**elay **C**hat
IND-CPA	**Ind**istinguishability under **C**hosen **P**laintext **A**ttack
KEP	**K**ey **E**xchange **P**rotocol
MAC	**M**essage **A**uthentication **C**ode
mpOTR	**m**ulti-**p**arty **O**ff **T**he **R**ecord
OTR	**O**ff **T**he **R**ecord
PGP	**P**retty **G**ood **P**rivacy
SHA	**S**ecure **H**ash **A**lgorithm
TOR	**T**he **O**nion **R**outer

List of Symbols

AES_{CTR}	AES block cipher in counter mode
$\mathcal{O}$	Security Adversary
$\mathcal{M}$	Privacy Adversary
$\mathcal{T}$	Consensus Adversary
H	A hash function
$\oplus$	The XOR operator
$\|$	The concatenation operator
$\odot$ or $\diamond$	A generic operator

Dedicated to the memory of Aaron Swartz

Chapter 1

Περιγραφή Πρωτοκόλλου

Εδώ θα παραθέσουμε μια περιγραφή του πρωτοκόλλου το οποίο υλοποιήσαμε.

1.1 Εισαγωγή

Το Pidgin είναι μια διαδεδομένη εφαρμογή desktop για συνομιλίες πραγματικού χρόνου. Συνοδεύεται από το OTR πρόσθετο το οποίο, χρησιμοποιώντας το OTR πρωτόκολλο [11] [2] [12], προσθέτει στο Pidgin τη δυνατότητα των από άκρο σε άκρο κρυπτογραφημένων συνομιλιών μεταξύ δύο ατόμων. Έτσι προσφέρει ασφαλείς συνομιλίες στις οποίες μόνο οι συνδιαλεγόμενοι μπορούν να διαβάσουν τα μηνύματα που ανταλλάσσονται, τα οποία είναι κρυφά ακόμα και στον πάροχο επικοινωνίας. Παρότι το ίδιο το OTR πρόσθετο προσφέρει συνομιλίες μόνο δύο ατόμων, τα υποβόσκωντα πρωτόκολλα συχνά παρέχουν "δωμάτια" πολλών χρηστών, όπου πολλοί μπορούν να συνομιλούν ταυτόχρονα μεταξύ τους. Μέχρι τώρα όσοι μιλούσαν σε τέτοιου είδους δωμάτια δεν απολάμβαναν τα πλεονεκτήματα της από άκρο σε άκρο κρυπτογράφησης.

Στόχος της εργασίας μας είναι η υλοποίηση μιας βιβλιοθήκης για ασφαλείς συνομιλίες μεταξύ πολλών ατόμων. Επιπρόσθετα υλοποιούμε κι ένα πρόσθετο για το Pidgin το οποίο χρησιμοποιεί αυτή τη βιβλιοθήκη έτσι ώστε να επιτρέπει τους χρήστες του Pidgin να συνομιλούν ασφαλώς σε ένα οικείο περιβάλλον.

Η δουλειά μας βασίζεται θεμελιωδώς στο mpOTR paper [7]. Ακολουθώντας τις συμβάσεις του OTR πρωτοκόλλου, ο όρος "ιδιωτικός" χρησιμοποιείται για να περιγράψει τις ιδιότητες των συνομιλιών της πραγματικής ζωής:

- Εμπιστευτικότητα
  Μόνο οι συμμετέχοντες μπορούν να διαβάσουν τα μηνύματα

- Αυθεντικοποίηση
  Οι συμμετέχοντες είναι βέβαιοι ότι πραγματικά μιλάνε σε αυτούς που νομίζουν ότι μιλάνε

- Διαψευσιμότητα
  Κανείς δε μπορεί να αποδείξει σε κάποιον που δε συμμετείχε στη συνομιλία, ότι κάποιο συγκεκριμένος συμμετέχοντας έλαβε μέρος στη συνομιλία αυτή

- Προώθηση Μυστικότητας
  Εάν τα μακροπρόθεσμα μυστικά ενός χρήστη εκτεθούν σε κάποιον επιτιθέμενο, τότε αυτός δε μπορεί να διαβάσει κανένα μήνυμα το οποίο στάλθηκε παλαιότερα

Όταν έχουμε να κάνουμε για συνομιλίες πολλών ατόμων, μια ακόμα ιδιότητα απαιτείται. Αυτή η ιδιότητα λέγεται συνέπεια περιεχομένων δωματίου, και γενικά δηλώνει ότι όλοι οι συμμετέχοντες έχουν την ίδια εικόνα για τα μηνύματα που έχουν σταλθεί σε κάποιο δωμάτιο.

Για να υλοποιήσουμε το mpOTR πρωτόκολλο το οποίο περιγράφεται στο [7], έπρεπε να συγκεκριμενοποιήσουμε τα υπο-πρωτόκολλα τα οποία χρησιμοποιούταν ως μαύρα κουτιά και δεν περιγράφηκαν πλήρως. Προτείνουμε μια συγκεκριμένη Διαψεύσιμη Ανταλλαγή Κλειδιών Υπογραφής (DSKE) η οποία βασίζεται σε εκτέλεση κατά ζεύγη του τριπλού Diffie–Hellman πρωτοκόλλου. Για την Ομαδική Συμφωνία Κλειδιού (GKA) χρησιμοποιούμε το πρωτόκολλο που περιγράφεται στο [9], αλλά χρησιμοποιούμε κλασσικό Diffie–Hellman (δηλαδή όχι Diffie–Hellman ελλειπτικών καμπυλών).

Υλοποιούμε την mpOTR βιβλιοθήκη ως κομμάτι της αρχικής OTR βιβλιοθήκης όπως φαίνεται στο το github repo μας[1], η οποία μέχρι τώρα πρόσφερε συνομιλίες μόνο για δύο συμμετέχοντες. Το πρόσθετο μας βασίζεται στο ήδη υπάρχον OTR πρόσθετο το οποίο αναπτύσσεται από την κοινότητα του OTR, και μπορεί κανείς να το δει στο το github repo μας[2].

1.2 Το Πρωτόκολλο

Στον αλγόριθμο 1 παρουσιάζουμε τη συμπεριφορά του πρωτοκόλλου. Το πρωτόκολλο χωρίζεται σε διάφορες φάσεις τις οποίες ονομάζουμε υπο-πρωτόκολλα. Τα τέσσερα πρώτα από αυτά (Offer, DSKA, GKA και Attest) είναι υπεύθυνα για να κατασκευάσουν όλη την απαραίτητη πληροφορία που απαιτείται ώστε να λάβει χώρα μια ιδιωτική συνομιλία. Το Communication υπο-πρωτόκολλο είναι αυτό το οποίο αναλαμβάνει να φέρει εις πέρας την ίδια τη συνομιλία. Τέλος το Shutdown υπο-πρωτόκολλο είναι υπεύθυνο ώστε να γίνει κάθε απαιτούμενη ενέργεια που πρέπει να συμβεί πριν κλείσει μια συνομιλία. Παρουσιάζουμε εν συντομία τα υπο-πρωτόκολλα αυτά παρακάτω.

Κατά τη διάρκεια του Offer υπο-πρωτοκόλλου, οι συμμετέχοντες υπολογίζουν ένα αναγνωριστικό sid για τη συνομιλία. Αυτό είναι ένας αριθμός, μοναδικός με μεγάλη πιθανότητα, που ταυτοποιεί τη συνομιλία.

Κατά το DSKE υπο-πρωτόκολλο, κάθε συμμετέχοντας κατασκευάζει έναν πίνακα αντιστοίχησης $\mathcal{S}$ ο οποίος αντιστοιχεί κάθε συμμετέχοντα σε ένα κλειδί υπογραφής το οποίο θα χρησιμοποιηθεί γι αυτή τη συνομιλία. Κάθε συμμετέχοντας παράγει ένα εφήμερο κλειδί υπογραφής με το οποίο θα αυθεντικοποιεί τα μηνύματά του. Έπειτα κάθε συμμετέχοντας στέλνει το δημόσιο κομμάτι του κλειδιού υπογραφής του με κάθε άλλο συμμετέχοντα, χρησιμοποιώντας μια Διαψεύσιμη Αυθεντικοποιημένη Ανταλλαγή Κλειδιού (DAKE). Όταν όλοι έχουν ανταλλάξει τα κλειδιά τους με όλους ο κάθε συμμετέχοντας έχει κατασκευάσει τον πίνακα αντιστοίχισης του. Αφού η ανταλλαγή κλειδιού είναι διαψεύσιμη, το ίδιο ισχύει και για τα κλειδιά υπογραφής. Θα μιλήσουμε πιο αναλυτικά για το DSKE και το DAKE στην παράγραφο (1.3.1).

Κατά το GKA υπο-πρωτόκολλο, οι συμμετέχοντες παράγουν ένα κοινό κλειδί $\mathcal{K}$ το οποίο θα χρησιμοποιηθεί για να παραχθούν κλειδιά κρυπτογράφησης. Τα κλειδιά αυτά θα χρησιμοποιηθούν για να κρυπτογραφηθούν τα μηνύματα που θα σταλούν κατά τη συνομιλία. Το υπο-πρωτόκολλο αυτό περιγράφεται αναλυτικότερα στην παράγραφο (1.3.2).

Κατά το Attest υπο-πρωτόκολλο οι συμμετέχοντες αυθεντικοποιούν το αναγνωριστικό sid και σιγουρεύονται ότι έχουν φτάσει στον ίδιο πίνακα αντιστοίχισης κλειδιών υπογραφής $\mathcal{S}$.

[1] https://github.com/Mandragorian/libotr/tree/mpotr

[2] https://github.com/Mandragorian/pidgin_otr/tree/mpotr_integration

Algorithm 1: mpOTR($\mathcal{P}$) — τρέχει μια συνεδρία του πρωτοκόλλου mpOTR
Input: $\mathcal{P}$: participants list
Result: Executes a run of the mpOTR protocol

```
begin
    sid ← Offer(P)
    S ← DSKE(P, sid)
    K ← GKA(P, sid, S)
    A ← Attest(P, sid, S)
    if A = ⊥ then
        return "Error"
    end
    T := Communication(P, sid, S, K)
    C ← Shutdown(P, sid, S, T)
    if ConsensusForAll(C) then
        return "OK"
    else
        return "Error"
    end
end
```

Κατά το Communication υπο-πρωτόκολλο, λαμβάνει χώρα η ίδια η συνομιλία. Οι χρήστες χρησιμοποιούν το κοινό μυστικό $\mathcal{K}$, τα εφήμερα κλειδιά υπογραφής και τον πίνακα αντιστοίχισης $\mathcal{S}$, ώστε να κρυπτογραφήσουν και να αυθεντικοποιήσουν τα μηνύματά τους. Όταν τελειώσει αυτή η φάση παράγεται ένα αντίγραφο της συνομιλίας το οποίο περιέχει όλα τα μηνύματα της συνομιλίας.

Κατά το Shutdown υπο-πρωτόκολλο, οι συμμετέχοντας αποφασίζουν αν υπάρχει συνέπεια περιεχομένων δωματίου και αποκαλύπτουν τα ιδιωτικά κομμάτια των κλειδιών υπογραφής τους. Εάν τα περιεχόμενα είναι όντως συνεπή τότε λέμε ότι υπάρχει ομοφωνία. Η αποκάλυψη των ιδιωτικών κλειδιών υπογραφής προσθέτει επιπλέον διαψευσιμότητα στο πρωτόκολλο, όπως και η αποκάλυψη των MAC κλειδιών στο OTR πρωτόκολλο. Παρόλα αυτά είναι προαιρετικό βήμα καθώς το πρωτόκολλο που προτείνεται είναι διαψεύσιμο και χωρίς την αποκάλυψη.

1.3 Τα υπο-πρωτόκολλα

Εδώ θα παρουσιάσουμε τα δύο υπο-πρωτόκολλα τα οποία δεν περιγράφονται στο [7], δηλαδή τη Διαψεύσιμη Ανταλλαγή Κλειδιών Υπογραφής και την Ομαδική Συμφωνία Κλειδιού.

1.3.1 DSKE

Στο[7] η ανταλλαγή κλειδιών υπογραφής περιγράφηκε χρησιμοποιώντας ένα πρωτόκολλο που το ονόμαζε ως Διαψεύσιμη Αυθεντικοποιημένη Ανταλλαγή Κλειδιού (DAKE) ως μαύρο κουτί. Στην υλοποίησή μας χρησιμοποιούμε το τριπλό Diffie–Hellman πρωτόκολλο ως DAKE, το οποίο είναι αυθεντικοποιημένο και διαψεύσιμο.

Κάθε συμμετέχοντας εκτελεί μια τριπλή Diffie–Hellman ανταλλαγή κλειδιού με κάθε άλλο συμμετέχοντα στο δωμάτιο και έτσι κατασκευάζουν ένα κοινό μυστικό. Με αυτό το μυστικό θα κρυπτογραφήσει και έπειτα θα αυθεντικοποιήσει το δημόσιο κομμάτι του κλειδιού υπογραφής του, και θα στείλει το αποτέλεσμα στον άλλον συμμετέχοντα.

Algorithm 2: SendUpflow($InterKeys$, x, $\hat{Y}$) — στέλνει την νέα λίστα ενδιάμεσων κλειδιών στον επόμενο συμμετέχοντα.

Input: $InterKeys$: previous intermediate key list
x : user's secret key
$\hat{Y}$: the next participant
Result: Sends the new intermediate key list to the next participant
begin
  $inter\_key\_list \leftarrow []$
  $inter\_key\_list.Append(InterKeys.Last())$
  foreach k *in* $InterKeys$ **do**
    $inter\_key\_list.Append(k^x)$
  end
  $AuthBroadcast(\hat{Y} \| inter\_key\_list)$
end

Αφού όλοι οι συμμετέχοντας έχουν ανταλλάξει τα κλειδιά υπογραφής τους με όλους τους άλλους, με τον τρόπου που περιγράφηκε παραπάνω, έχουν σχηματίσει τον πίνακα αντιστοίχισης $\mathcal{S}$. Είναι άξιο να σημειωθεί ότι η DSKE είναι η μόνη φάση κατά την εγκατάσταση της συζήτησης κατά την οποία $O(n^2)$ μηνύματα ανταλλάσσονται. Μετά στέλνονται $O(n)$ μηνύματα.

Μια σχηματική περιγραφή του πρωτοκόλλου φαίνεται στο σχήμα 5.1.

1.3.2 GKA

Για την Ομαδική Συμφωνία Κλειδιού χρησιμοποιούμε το πρωτόκολλο που περιγράφεται στο [9]. Επαναλαμβάνουμε ότι η βασική ιδέα είναι η Diffie–Hellman ανταλλαγή κλειδιού, γενικευμένη για πολλούς συμμετέχοντες.

Κατά τη GKA τα μηνύματα ανταλλάσσονται σε δύο φάσεις. Στη φάση της ροής και στη φάση της αντιροής. Κατά τη φάση της ροής τα μηνύματα ανταλλάσσονται σειριακά μεταξύ των συμμετεχόντων. Ο κάθε συμμετέχοντας υπολογίζει κάποια ενδιάμεσα κλειδιά, βασιζόμενος στα ενδιάμεσα κλειδιά που έλαβε από τον προηγούμενό του. Τα ενδιάμεσα κλειδιά αυτά θα τα στείλει στον επόμενο του. Μετά το πέρας της φάσης ροής, ο τελευταίος συμμετέχοντας έχει αρκετά δεδομένα ώστε να παράξει το κοινό μυστικό. Επίσης έχει όλη την πληροφορία που χρειάζεται και από τους υπόλοιπους συμμετέχοντες, ώστε να φτιάξουν και αυτοί το κοινό μυστικό. Αυτή η πληροφορία μεταδίδεται στους υπολοίπους όπως φαίνεται στο σχήμα 5.3.

Στους αλγορίθμους 2, 3, και 4 παρουσιάζεται η κεντρική ιδέα της GKA.

1.4 Πρωτογενείς Διαδικασίες

1.4.1 Ομάδα Diffie–Hellman

Στην υλοποίηση μας επαναχρησιμοποιήσαμε τον κώδικα για την ανταλλαγή κλειδιού Diffie–Hellman από τη βιβλιοθήκη libotr. Αυτό σημαίνει ότι χρησιμοποιούμε κλασσικό Diffie–Hellman και συγκεκριμένα την ομάδα υπ. αριθμόν 5 με συντελεστή μήκους 1536 bit. Στους αλγορίθμους που περιγράψαμε παραπάνω όλες οι υψώσεις σε εκθέτη γίνονται σε αυτή την ομάδα.

Algorithm 3: SendDownflow($InterKeys$, x) — εκπέμπει την λίστα ενδιάμεσων κλειδιών αντιροής στους υπόλοιπους συμμετέχοντες.

Input: $InterKeys$: previous intermediate key list
x : user's secret key
Result: Broadcasts the downflow intermediate key list to the other participants

```
begin
    inter_key_list ← []
    foreach k in InterKeys do
        inter_key_list.Append(k^x)
    end
    AuthBroadcast(inter_key_list)
end
```

1.4.2 Κρυπτογράφηση

Για την κρυπτογράφηση χρησιμοποιούμε AES-128 σε Counter τρόπο λειτουργίας, όπως και στο απλό OTR. Επιλέξαμε τον AES με κλειδί 128 bit και όχι με 256 αφενός γιατί η ομάδα Diffie–Hellman που χρησιμοποιούμε δεν παρέχει 256 bit εντροπίας και αφετέρου εξαιτίας διαφόρων μελετών που υποδεικνύουν ότι ο αλγόριθμος δρομολόγησης κλειδιού του AES-128 είναι πιο ανθεκτικός σε επιθέσεις [1] [4].

Για να κρυπτογραφηθεί ένα μήνυμα, ο χρήστης παραθέτει το κοινό μυστικό με το προσωπικό του αναγνωριστικό για τη συζήτηση και δημιουργεί ένα προσωπικό κλειδί ως εξής:

$$k_{enc} = H(id_{\pi\rho o\sigma\omega\pi\iota\kappa o}||master\ key)$$

Για τον μετρητή ο κάθε χρήστης διατηρεί τοπικά το δικό του προσωπικό πάνω μισό (τα 8 πιο σημαντικά bytes) το οποίο αυξάνει κατά 1 κάθε φορά που στέλνει ένα μήνυμα. Το κάτω μισό (8 λιγότερο σημαντικά bytes) είναι πάντα αρχικοποιημένα στο 0. Σε κάθε μήνυμα που στέλνεται προστίθεται το πάνω μισό του μετρητή. Το κρυπτοκείμενο παράγεται ως εξής (όπου ctr είναι το πάνω μισό του μετρητή):

$$ciphertext = AES_{CTR}(k_{enc}, ctr||0, plaintext)$$

Για να αποκρυπτογραφηθεί ένα μήνυμα, ο χρήστης παραθέτει το κοινό μυστικό με το προσωπικό id του αποστολέα του μηνύματος.

$$k_{dec} = H(id_{sender}||master\ key)$$

Και με τον μετρητή που υπάρχει στο μήνυμα αποκρυπτογραφεί ως εξής:

$$plaintext = AES_{CTR}(k_{dec}, ctr||0, ciphertext)$$

Αυτό το σχήμα κρυπτογράφησης χρησιμοποιείται ώστε να μην υπάρχει πιθανότητα να επαναχρησιμοποιηθεί κάποια δυάδα μετρητή-κλειδιού κρυπτογράφησης, κάτι που ο Counter τρόπος λειτουργίας απαιτεί να μη συμβεί.

Algorithm 4: GKA($\mathcal{P}$, sid, $\mathcal{S}$) - εκτελεί μια Ομαδική Συμφωνία Κλειδιού και παράγει το κοινό μυστικό στο πλαίσιο του συμμετέχοντα $\hat{X}$.

```
Input: 𝒫 : participants list
       sid : the session ID
       𝒮 : association table
Output: 𝒦 : the shared secret
begin
    x ← GenerateKey()
    Ŷ_prev ← X̂.Previous()
    Ŷ_next ← X̂.Next()
    if Ŷ_prev = NULL then                                   /* X̂ is first */
        SendUpflow([G], x, Ŷ_next)
    else
        repeat                                 /* wait for previous upflow */
            (Ŷ, R̂‖key_list) ← AuthReceive(𝒮)
        until R̂ = X̂;
        if Ŷ ≠ Ŷ_prev ∨ R̂‖key_list =⊥ then
            return error
        end
        if Ŷ_next ≠ NULL then                  /* X̂ is not first or last */
            SendUpflow(key_list, x, Ŷ_next)
        else                                                 /* X̂ is last */
            final_key ← key_list.Last()
            𝒦 ← final_key^x
            SendDownflow(key_list, x)
            return 𝒦
        end
    end
    repeat                                           /* wait for downflow */
        (Ŷ, key_list) ← AuthReceive(𝒮)
    until Ŷ = 𝒫.Last();
    if key_list =⊥ then
        return error
    end
    pos ← 𝒫.IndexOf(X̂)
    final_key ← key_list.Reverse().Get(pos)
    𝒦 ← final_key^x
    return 𝒦
end
```

1.4.3 Αυθεντικοποίηση

Για τις υπογραφές χρησιμοποιούμε τον αλγόριθμο EdDSA πάνω στην καμπύλη Ed25519 Κάθε μήνυμα υπογράφεται στο σύνολο του. Επιλέχτηκε αυτός ο αλγόριθμος για τη γρήγορη παραγωγή κλειδιού αλλά και για το μικρό μήκος υπογραφής. Αυτό σημαίνει ότι η υπογραφή καλύπτει τόσο τα δεδομένα όσο και τα μέταδεδομένα του μηνύματοσ, όπως το αναγνωριστικό της συνομιλίας, την τιμή του μετρητή και άλλα.

Chapter 2

Introduction

2.1 Motivation

Not much time has passed since Edward Snowden revealed the plans that a certain intelligence agency has for the internet. While the world had always been suspecting that the various 3-letter agencies had the capability of controlling the network at a large scale, everybody was shocked with the confirmation of those suspicions.

In a world where the need for easy instant communication must overcome the threat of constant surveillance, end-to-end encryption has become a necessity. It's not a coincidence that digital privacy has come into focus during the last years. One after another companies advertise the utilization of encryption in their products. Apparently, Instant Messaging is the most favorable means of communication when it comes to end-to-end encryption.

One of the oldest and commonly used protocols that provides privacy in Instant Messaging is the Off-The-Record (OTR). OTR was initially introduced in a paper named "Off-the-Record Communication, or, Why Not To Use PGP" in 2004 [11] and later improved in [2]. It was named after the homonymous method of journal sourcing. The primary motivation behind OTR was to provide deniable authentication for the conversation participants while keeping conversations confidential, as in real-life private conversations. The protocol is implemented as a C library and a Pidgin plugin, a user study of which can be found in [12].

Unfortunately, OTR does only apply in a two-party setting, where only two participants are exchanging messages. However, multi-party chat rooms are also very prominent in everyday communications. A protocol providing the same privacy properties as OTR in a multi-party setting was theoritically described in the "Multiparty Off-the-Record Messaging" paper by I. Goldberg et al. in 2009 [7]. This protocol is called multi-party OTR (mpOTR). Although it's been around since 2009, no actual implementation of mpOTR existed until now.

2.2 Our Contributions

The existing theoretical introduction of mpOTR protocol in [7] treats the underlying subrotocols as black boxes and does not describe them in detail. More specifically, two underlying subprotocols are left unspecified, namely the deniable Authenticated Key Exchange (denAKE) and the Group Key Agreement (GKA). These subprotocols play a key role in setting up the parameters needed for authentication and encryption.

We propose a full construction for the mpOTR Protocol. We specify every underlying sub-protocol. We also specify all the primitive algorithms used for several cryptographic functions, such as hashing, signing and encrypting. Finally, we propose a detailed low-level description of the protocol, including message structures, encoding, etc.

In addition, we provide the first implementation of mpOTR. Our implementation is a production-grade extension of the existing OTR library accompanied by a pidgin plugin, all writen in C. Both are open source projects, available in our github repositories[1][2]. We engineered our implementation in such a way that its security is easily reviewable, and, at the same time, facilitates the free software development model, where contributions in the source code are made from several independent authors.

2.3 Outline

In Chapter 3 we introduce some basic theoretic background. All the concepts, cryptographic primitives, and ideas presented there are essential building blocks of the mpOTR Protocol construction proposed in this thesis.

In Chapter 4 we specify the threat model of the mpOTR Protocol. We describe the different types of adversaries along with their goals. Then, we describe the goals of the mpOTR Protocol to achieve security against each type of adversary.

In Chapter 5 we present our mpOTR Protocol construction. First, we specify the desirable privacy properties of mpOTR conversations and the underlying network setting. Then, we present a high-level overview of the protocol followed by a detailed description of every building block. Afterwards, we specify several technical details. Finally, we specify the exact structure of the messages exchanged in mpOTR.

In Chapter 6 we present our actual implementation of mpOTR. We introduce several design challenges and the relevant decisions. Then we present the design model of the mpOTR library. Finally, we specify the Application Programming Interface that our implementation offers to the IM applications in order to utilize mpOTR in group conversations.

In Chapter 7 we present our modifications of the OTR pidgin plugin. We introduce the Graphical User interface and the workflow of a private group conversation.

In Chapter 8 we present other protocols that utilize end-to-end encrpytion in multiparty context.

In Chapter 9 we present several problems regarding specific parts of our mpOTR construction. While our construction is fully functional and secure, solving these problems would enhance the protocol's privacy and/or usabilty. We fully describe each of them and suggest possible solutions.

[1]https://github.com/Mandragorian/libotr
[2]https://github.com/Mandragorian/pidgin_otr

Chapter 3

Theoretic Background

In this chapter we will present some basic theoretic background. All the concepts, cryptographic primitives, and ideas presented here are essential building blocks of the protocol proposed in this thesis. As a result before someone carries on forward in this document, she should first have a basic understanding of this chapter.

3.1 Symmetric Encryption

The idea of symmetric encryption is quite simple. We suppose that there exists an algorithm called E, which takes two inputs. One input is a secret, called the key, and the other is some data we would like to encrypt.

Another algorithm called D again takes two inputs and is the reverse of E. This means that if we call D with inputs the output of E under some key k, and k itself, the output will be the plaintext data. This is shown in the below equation:

$$m = D(k, E(k, m))$$

3.2 Block Ciphers

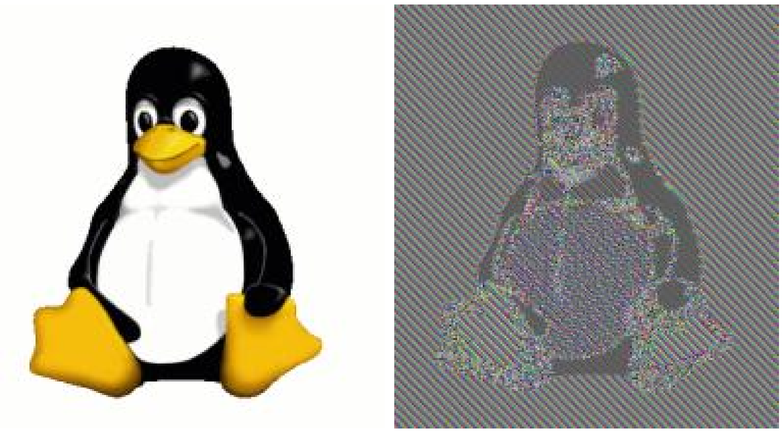
Block ciphers are a special case of symmetric encryption algorithms. What makes them special is that they operate on a constant length block of data, hence the name. While this constrain may appear very limiting, we will see that this is not the case. In fact block ciphers have dominated the field of symmetric encryption.

The block cipher which is most commonly used is called AES, also known as Rijndael. Its block size is 128 bits and depending on the AES version it has a key size of either 128, 192, or 256.

Exactly because AES is the most commonly used block cipher, it is also the most scrutinized and studied one. Thus the crypto community is quite confident that AES is a secure construction, fully capable to be used for encrypted communications.

3.3 Modes of Operation

As we already stated, block ciphers operate on fixed chunks of data. To overcome this limitation we need a mechanism so that we can break the plaintext in chunks of the cipher's block size, and then somehow apply the cipher in each chunk. There are many modes of operation out there, but we will only talk about two of them.



(α′) Original image (β′) Encrypted using ECB mode

FIGURE 3.1: ECB in practice. The original image was created by Larry Ewing using GIMP.

One is called Electronic Code Book (ECB), and it is the naive solution that one can come up with when first tackling the problem at hand. Suppose that we have a message M which is a multiple of the block size. We can then divide it into n chunks m_i, each being one block in length:

$$M = m_1 \| ... \| m_n$$

Then we encrypt each block using the bloc cipher and the same key such that we get ciphertext C:

$$C = c_1 \| ... \| c_n = E(k, m_1) \| ... \| E(k, m_n)$$

It is easy to see that the decryption is trivial:

$$M = D(k, c_1) \| ... \| D(k, c_n)$$

However, this has a crucial weakness. You can easily see that if two chunks are the same then the resulting ciphertext will also be the same. As a result data patterns in the plaintext may remain in the ciphertext as well. This might not seem important, but, believe us, it is detrimental to the mode's security. Don't believe us yet? A striking example is shown in figure 3.1 where ECB mode is used to encrypt a bitmap image which uses large areas of uniform colour.

The other is called Counter mode (CTR). In this mode the block cipher itself does not encrypt the plaintext. Instead it is used to produce a pseudo-random sequence of bits, in chunks equal to one block, which is then xor'ed with the plaintext.

To produce the i-th chunk s_i, it "encrypts" the number i with the provided key:

$$s_i = E(k, i)$$

As a result, a bit stream is generated as shown below:

$$S = s_1 \| s_2 \| ... \| s_n$$

And the resulting ciphertext is:

$$C = m_1 \oplus s_1 \| \ldots \| m_n \oplus s_n$$

To decrypt, one can generate the same bit stream and xor it again with the ciphertext:

$$M = c_1 \oplus s_1 \| \ldots \| c_n \oplus s_n$$

Notice that this means that we use the encryption algorithm of the block cipher both for encryption and decryption.

Counter mode is a secure mode of operation suitable for use in production. It also has a nice property called malleability, but more on that later.

3.4 Message Integrity

We now know how to keep a message secret. Another major cryptographic problem is that of message integrity. How can we be sure that a message we are reading came from who we think it did?

But why is this even necessary? Since only the sender and the receiver have the secret key, only they should be able to construct valid ciphertexts.

Well, in general this is not correct. Let's consider again the case of a block cipher used in CTR mode. Remember that in this mode the block cipher is not directly used for encryption. Instead it is used to generate a pseudorandom sequence of bits. This sequence is then xor'ed to the plaintext and the result is the ciphertext.

Now let's see what will happen if a bit of the ciphertext is flipped, and we try to decrypt it. Suppose m_i and c_i is the i-th bit of the plaintext and ciphertext respectively. s_i is the i-th bit of the pseudorandom sequence produced during the CTR mode encryption process. The follow relation holds:

$$m_i = c_i \oplus s_i$$

Flipping a bit means taking its complement. This means that the new plaintext bit M_i will be:

$$M_i = c_i{\prime} \oplus s_i = (c_i \oplus s_i){\prime}$$

This means that an attacker could actually produce valid ciphertexts for plaintexts of his choosing by tampering with already sent messages! This property is called malleability.

There are many more ways that an attacker could tamper with sent messages besides utilising the malleability properties of an encryption scheme. This means that we need ways to authenticate our messages before we sent them.

3.5 Hash Functions

Before we tackle that problem we will have a look at hash functions. These hash functions are algorithms that accept as input some data of arbitrary length and output some value of fixed length.

If this value is computed in such a way that the hash function H satisfies the following properties (as stated in [10]), then we call the function a *cryptographic* hash function.

- Pre-image resistance

  For essentially any output value h of the hash function H it is computationally infeasible to find any input x such that $H(x) = h$.

- Second pre-image resistance

  For a given input x it is computationally infeasible to find a second input $x\prime$ such that $x \neq x\prime$ and $H(x) = H(x\prime)$.

- Collision resistance

  It is computationally infeasibly to find any pair of inputs x and $x\prime$ such that $H(x) = H(x\prime)$[1].

Notice that collision resistance implies second pre-image resistance. Likewise second pre-image resistance implies pre-image resistance.

3.6 Message Authentication

For the purpose of message authentication there are two main categories of solutions. The first category uses a shared secret to create some sort of electronic signature for some data that we want to authenticate. This type of signature is called a Message Authentication Code.

The second category uses a pair of keys, the private key, known only to the person signing the data, and the public key, known to everybody. The public key is used to verify the signature.

3.6.1 Message Authentication Codes (MAC)

This case has many similarities with symmetric encryption. Again we assume that there is a secret value, called the key, known only to the two parties communicating. We also assume that an algorithm called MAC exists which produces the signature, called the tag or mac.

This algorithm takes two inputs, the secret key k and the message[2] m itself. The algorithm works in a way such that the produced tag t can only be calculated if you know k.

$$t = MAC(k, m)$$

When the sending party wants to transmit a message, she calculates the tag and appends it to the message so that she sends $m \| t$. The receiving party can recalculate the tag on his own. He then checks if the appended tag is the same with the tag that he calculated. If the two tags are the same then he accepts the message, otherwise he rejects it as it is probably tampered with.

[1] This and the previous property might seem the same, but notice that in the Second pre-image property x is fixed, while in the collision resistance property it is not

[2] This message can be plaintext or already encrypted

3.6.2 Public key signatures

In this case we have two distinct algorithms. On is called $Sign$ and is used to produce the signatures. The other is called $Verify$ and is used to verify data.

The sender generates a tag t using the $Sign$ algorithm:

$$t = Sign(priv, m)$$

where $priv$ is the private key. He then appends the tag to the message as in the MAC case and transmits $m \| t$.

The receiving party then uses the $Verify$ algorithm. This algorithm accepts as inputs the message m, the tag t, and the public key pub. Its result is boolean, responding ”YES” if the message is verified and ”NO” if it is not:

$$result = Verify(m, t, pub)$$

if $result$ is ”YES” then she accepts the message. Otherwise she rejects it.

3.7 Key Exchange Protocols (KEP)

The goal of any Key Exchange Protocol is to allow two users to agree on a secret value, known only to them, by exchanging some publicly known information. This might be counter intuitive at first but as we will see in the next section this goal can be achieved with a very simple construction.

Basically a Key Exchange Protocol is any mechanism that provides two functions. One of them returns a tuple containing some private and some public information. The public information returned by that function must be sent to the other user so that he can calculate the secret value. The other function accepts the public values of the other user and the private values of this user and returns the secret value.

$$< priv, pub > = genkey()$$
$$secret = calculate\_secret(priv_i, pub_j)$$

Figure 3.2: The KEP interface. pub_j is the public information of user j and $priv_i$ the private information of user i.

3.8 Diffie–Hellman key exchange

The Diffie–Hellman key exchange is, as the name suggests, a key exchange protocol. It plays a major role in our proposed protocol, since it is a building block of many of its components. It was introduce by Whitfield Diffie and Martin Hellman in [14].

In this section we will examine how this KEP is constructed, and what public values must be exchanged.

First, we remind to the readers the operation of multiplication modulo a number, $\odot$. We want to multiply two numbers a and b modulo a number n, called ”the modulo”. This means that we first multiply the two numbers as usual. Then we calculate the

---

Algorithm 5: The *genkey* function

Result: The private and public information needed by the protocol
begin
  $x \leftarrow random\_in\_range(1, p-1)$
  $p \leftarrow g^x$
  return (x, p)
end

---

remainder of the result when it is divided by the modulo. This remainder is the result of the multiplication of those two numbers a, b modulo n. This means that if:

$$ab = np + r$$

then:

$$a \odot b = r$$

In general we will abuse the notation of exponentiation and symbolize:

$$a^x \equiv a \odot \cdots \odot a = a^x \mod p$$

These are all the maths needed to understand how the Diffie–Hellman key exchange works. To understand why it is also secure is a whole different matter and we will not cover it in this publication.

The Diffie–Hellman construction supposes that the two users already agree on two values g and p which are publicly known. The number p must be prime and is called "the modulo" of the protocol. The number g is called the "generator" and has the property that for every number $k \in [1 \ldots p-1]$ there exists a number l in the same range such that $g^l = k$.

The private information for a user is any random integer x such that $1 < x < p$. The public information is $g^x \mod p$ (any exponentiation from now on will be modulo p).

The calculation of the shared secret is trivial. A user i, with private information x, and public g^x, receives the public information g^y of another user j. He then calculates the value $s = (g^y)^x$ which is the shared secret. Now note that with i's public information, j can also calculate the same value $s = (g^x)^y$. The calculated value is the same for the two users since:

$$(g^y)^x = (g^x)^y = g^{xy}$$

In algorithm 5 we see the *genkey* function, and in algorithm 6 the *calculate_secret* function.

From now on, the public and private information of a user will be called public and private keys accordingly.

Algorithm 6: The *calculate_secret* function

Input: g^y: the public value of the other user, x: the secret value of the user calling the function

Result: The shared secret, which is known only to the two users

begin

  return $(g^y)^x$

end

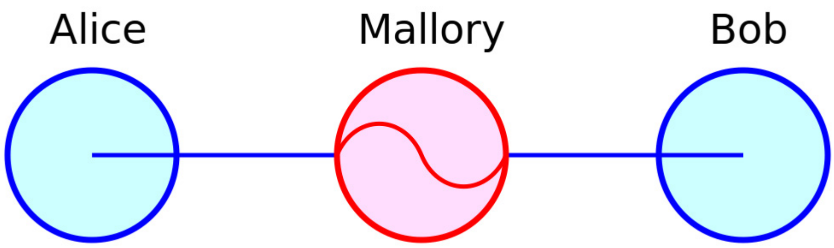


FIGURE 3.3: This figure demonstrates the situation just after Mallory has performed her Person-in-the-Middle attack.

3.9 Person-in-the-Middle attacks

Although Diffie–Hellman is a great protocol for calculating shared secrets, it has a grave disadvantage. Consider the following scenario, where Mallory, an evil attacker can control the network so that she can read, drop, and inject packets:

Alice wants to privately communicate with Bob. She generates a private key x and sends the public key g^x to Bob.

Mallory interjects and copies and then drops the packet containing Alice's public key. She generates a private key $x\prime$ and the public counterpart $g^{x\prime}$. She performs a Diffie–Hellman exchange with Alice's key and calculates $s_1 = g^{xx\prime}$, since she knows $x\prime$. She then sends her public key to Bob.

Bob believes that the public key he just received belongs to Alice. He generates his private key y, and sends the public key g^y to Alice. He also calculates $s_2 = g^{yx\prime}$ what he thinks is a shared secret known only to him and Alice.

Mallory interjects again to copy and drop the just sent package from the network. She calculates $s_2 = g^{yx\prime}$, again she knows $x\prime$. Then she sends her public key $g^{x\prime}$ to Alice, posing as Bob.

Now Alice receives a public key which she thinks belongs to Bob. Like Bob she calculates the shared secret $s_1 = g^{xx\prime}$ which she thinks is only known between her and Bob.

The situation now is that Alice and Bob use two different secrets to communicate which are both known to Mallory. Mallory can decrypt any message sent by Alice, for example, since she knows s_1. She then can re-encrypt it using s_2 and relay it to Bob.

Neither Alice nor Bob will be able to know that such tampering is taking place and will continue to communicate, thinking that everything is fine.

The above scenario was described with the Diffie–Hellman key exchange in mind. However any similar situation where a malicious attacker can get between two communicating partners who think they are talking directly to each other is known as a Person-in-the-Middle attack.

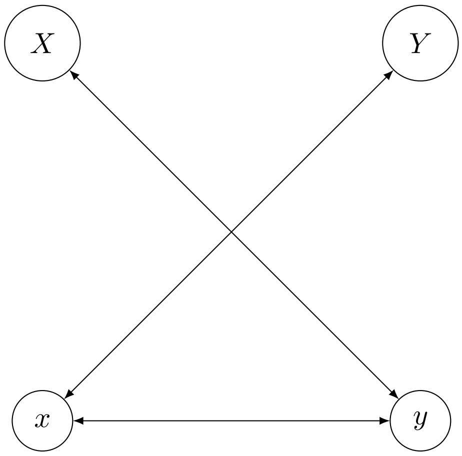


FIGURE 3.4: These are the 3 Diffie–Hellman key exchanges that are performed in this protocol

3.10 Triple Diffie–Hellman key exchange

Triple Diffie–Hellman is another key exchange protocol. It is an improvement of Diffie–Hellman which introduces negligible overhead in the complexity of the protocol. With doubling length of the first message containing the public keys of a user, this protocol provides both authentication and deniability.

3.10.1 An Overview

In this key exchange each user has a Diffie–Hellman longterm keypair. This should be authenticated to other users to exclude a person-in-the-middle attack, like any other public key scheme.

When Alice wants to communicate with Bob, she generates an ephemeral Diffie–Hellman key. Suppose X is her longterm private key and x is her ephemeral private key.

She then sends the tuple (g^X, g^x) to Bob, who upon receiving it does exactly what Alice did. He generates an ephemeral key y. Supposing Y is his longterm private key, he then calculates the shared secret s as shown below:

$$s = g^{Xy} \| g^{xY} \| g^{xy}$$

Notice that an asymmetry exists, in the above calculation. Should $g^X y$ got first or should g^{xY}? This can easily be decided by, for example, comparing the longterm public keys. If $g^X > g^Y$ then:

$$s = g^{Xy} \| g^{xY} \| g^{xy}$$

and if not:

$$s = g^{xY} \| g^{Xy} \| g^{xy}$$

And both users will calculate the same secret.

The shared secret generated by the above protocol is forward secret, and authenticated, yet deniable.

It is forward secret since even if someone forces Alice or Bob to hand over their long term secrets, the shared secret cannot be reconstructed. This is true because to calculate the shared secret, either x or y are needed, and we assume that ephemeral secrets are destroyed after a session finishes.

It is authenticated since to calculate the above secret one needs to know either X or Y.

It is deniable since the only values exchanged during the protocol are public keys.

3.10.2 The catch

There is a catch however. Suppose Eve would like to be able to read *some* message that Alice has authored and sent to Bob. What she can do is create an "ephemeral" key e. She then starts a conversation with Alice, posing as Bob. She can do this since Y is a longterm key, and g^Y may have previously been publicly exchanged. Alice might respond, create the shared secret and start communicating. Eve does NOT know the shared secret, so she can't read any messages or prove anything, but she might be able to calculate it in the future.

If Eve at some point in the future forces Bob to hand over his long term secrets then she can recreate the shared secret. She can thus read the first message that Alice has tried to send to "Bob" using the shared secret calculated with the phony ephemeral. This is possible since she now knows Y (Bob's longterm secret) and she also knows e. This breaks the forward secrecy of the protocol.

This can be easily solved if both Alice and Bob first make sure that they have both arrived at the same shared secret.

Suppose that Alice has calculated the shared secret s. She uses that secret to send an encrypted and authenticated message to Bob. That message contains data known to everybody, the word "confirmation" for example. She won't send any further messages unless she receives the same confirmation message from Bob.

On receiving the confirmation message Bob can be sure that Alice has calculated the shared secret. He then sends a confirmation message himself, so that Alice too can be sure that Bob has calculated the shared secret. Eve could not have sent such a message because she currently doesn't know Y, needed for her to calculate s.

3.11 Message Ordering

One difficulty that multi-party chat protocols need to overcome is that of message ordering. This is confusing at first. Why would the ordering of the messages be so important? If a conversation is reordered it shouldn't make any sense.

Unfortunately this is not the case. To illustrate why message order is important we will examine the so called "ice cream attack".

Suppose that Alice, Bob, and Mallory are chatting in one chat room. Mallory wants to make Alice believe that Bob is a dangerous criminal. We assume that she has significant control over the network and can delay packages. To achieve her goal she does the following:

She sends the message "Who wants some ice cream?" in the chat room. But she delays the package going to Alice, so that Alice does not receive it yet. Bob will obviously reply that he wants ice cream. Let's suppose he sends the message "I do". Again Mallory will put the package on hold before she delivers it to Alice. And finally she transmits the message "Who wants to do something illegal?", which she allows to be delivered to both Alice and Bob. She then allows Bob's message, saying

"I do", to go through to Alice. After that she also allows her original message, saying "Who wants some ice cream?" to also go to Alice.

Now, what did Alice actually see? First, she saw that Mallory asked who wanted to do something illegal. And then she saw Bob saying that he does! In her eyes, Bob is willing to do engage in illegal activities. Maybe she shouldn't hang out with him any more.

And that exactly is the "ice cream attack". Can multi-party chat protocols defend against such vulnerabilities? The problem at hand is generally still open and goes beyond the scope of this publication. However at section 9.4 we shall briefly examine a proposed solution which is compatible to our protocol.

Chapter 4

Threat Model

Before introducing our mpOTR construction we should define the threat model. We adopt the threat model specified in [7]. First, we introduce the three different types of adversaries. Then, we describe the goals of the mpOTR Protocol regarding each adversary.

4.1 The Adversaries

4.1.1 Security adversary $\mathcal{O}$

This adversary's goal is to read the messages of the chatroom. Let $T_c^{\hat{X}}$ be the transcript of chatroom c owned by participant $\hat{X}$. Then, $\mathcal{O}$ is successful, if he can read any message from transcript $T_c^{\hat{A}}$, where $\hat{A}$ is an honest participant, without receiving it from transcript $T_c^{\hat{X}}$ for any participant $\hat{X}$ where $\hat{X} \neq \hat{A}$. While $\mathcal{O}$ can, both passively and actively, control the network, decrypt messages sent in other chatrooms and even participate in other sessions, he has limited access on the room he wants to attack. Not only can he not participate in the session under attack, but he also cannot ask for any secret shared between the participants of the specific room. He has the ability to inject messages of his liking in the chatroom by asking an, otherwise honest, user. In essence $\mathcal{O}$ is a somewhat formal definition of the notion of IND-CPA attacks in the multiparty setting.

4.1.2 Privacy adversary $\mathcal{M}$

The privacy adversary aims to break the deniability of the protocol. He is successful if he can prove to a judge $\mathcal{J}$ that a user $\hat{A}$ participated in, read messages from or authored messages in chatroom c.

His restrictions are very few. He can collaborate with $\mathcal{J}$ before the creation of c, participate fully in c and even force $\hat{A}$ to reveal his long term secrets in front of the judge.

4.1.3 Consensus adversary $\mathcal{T}$

Definition of Consensus

For two participants $\hat{A}$ and $\hat{B}$, consensus is reached on $T_{C_1}^{\hat{A}}$ when $\hat{A}$ believes $\hat{B}$ claims to have a transcript $T_{C_2}^{\hat{B}}$ such that:

- C_1 has the same set of participants as C_2;
- C_1 and C_2 are the same chat room instance;
- $T_{C_2}^{\hat{B}}$ has the same collection of messages as $T_{C_1}^{\hat{A}}$;
- $T_{C_2}^{\hat{B}}$ and $T_{C_1}^{\hat{A}}$ agree an each message's origin.

Notice that the above definition is not symmetric. This means that $\hat{A}$ can reach consensus with $\hat{B}$ without necessarily $\hat{B}$ reaching consensus with $\hat{A}$.

The interpretation of the term "collection of messages" is intentionally left unclear. This way, each application can handle the ordering of the messages in different ways.

$\mathcal{T}$'s goal

$\mathcal{T}$ is successful when he is able to force an honest user $\hat{A}$ to believe that consensus is reached with another honest user $\hat{B}$ when at least one condition from 4.1.3 does not hold. Notice that only $\hat{A}$ and $\hat{B}$ must be honest. $\mathcal{T}$ can otherwise control other users as he sees fit.

4.2 The Goals of the Protocol

The protocol must provide some defence mechanisms against all of the above.

The security adversary $\mathcal{O}$ can in no way be successful. The protocol must ensure that no one outside a chat room can read messages authored for it. This would be a catastrophic failure.

To defend against the privacy adversary $\mathcal{M}$ is also crucial. One might think, that since an attacker can not read messages, it won't make much difference if they are not deniable. However there are many situations that even evidence that you talked to someone can be incriminating.

If, for example, Ed wanted to reveal to a journalist some evidence about the wrongdoings of a state organization and did that in a non deniable manner then he would be busted. The correlation between him contacting the journalist and the subsequent release of the information he revealed would mark him as a whistle blower.

Lastly, the consensus adversary $\mathcal{T}$ is not strictly cryptographic but is of no less importance. If an attacker could manipulate the transcripts of two different users, but then convince them that they have received the same messages he could have quite some power over them. Every multi-party chatting protocol must ensure that all participants view the same messages[1].

[1]The order of the messages is also very important but this is an open problem and, while various solutions have been proposed, it is not addressed by this work. See section 3.11 for a theoretic approach and section 9.4 for our proposed solution.

Chapter 5

The mpOTR Protocol

5.1 Properties of private conversations

Following the conventions of the OTR protocol, the term ”private” is used to describe the properties of casual real-life conversations. The following four properties are both required in two-party as well as multi-party private conversations:

Confidentiality No one else apart from the chat room participants can read the messages exchanged.

Authentication You are assured that the participants are who you think they are.

Repudiation The messages sent do not have digital signatures that are checkable by a third party. Anyone can forge messages after a conversation to make them look like they came from you. However, during a conversation, other participants are assured the messages they see are authentic and unmodified.

Forward secrecy If you lose control of your private keys, no previous conversation is compromised.

In the context of the multi-party chat room, one more property is required:

Chat room transcript consistency All participants share the same view over the messages exchanged in a given chat room.

5.2 Underlying network setting

The mpOTR Protocol is designed to run on top of an existing network setting. This may be any application layer protocol, such as Jabber/XMPP, IRC, etc.

We assume that the application layer protocol offers the following network primitives in the context of chatrooms:

- $Broadcast(M)$ — sends a message M over the chat room where it can be $Receive()$'ed by all other participants.
- $Receive() \to (\hat{A}, M)$ — returns any waiting message M received by the participant that invokes $Receive()$ along with M's alleged author $\hat{A}$.

In the following, we use M as a representation of either a single value, or multiple values which we denote using the concatenation symbol ($\|$). We describe the exact structures and value encodings of mpOTR messages in section 5.9.

We also assume that any message fragmentation is handled by the application layer protocol, which delivers mpOTR messages as whole to the mpOTR Protocol. However, in some cases this may not be the case. We discuss this in more detail in section 9.1.

Finally, notice that we assume no security provided by the underlying network. An adversary may have complete control over the network and may modify, drop and/or deliver messages at will. The realization of the privacy properties exclusively relys on the mpOTR Protocol.

5.3 High level protocol overview

In Algorithm 7 we illustrate a high-level overview of a single session of the mpOTR Protocol. The whole protocol has been divided into sequential phases, which we call sub-protocols. The first four of them (Offer, DSKE GKA and Attest) are responsible for setting up all the needed parameters, for the private communication to take place. The Communication sub-protocol is the one that governs the actual private group conversation. Finally, the Shutdown sub-protocol is responsible for every action that needs to be done before ending each private session. We briefly describe the function of each sub-protocol below.

During the Offer sub-protocol, the participants create a Session ID, sid. This is a unique (with high probability) number identifying the session. The Offer sub-protocol is described in more detail in section 5.4.1.

During the DSKE sub-protocol, every participant creates an association table $\mathcal{S}$ which maps each participant $\hat{X}$ to the public part of the signing key $E_{\hat{X}}$ he is going to use for this session. Each participant generates an ephemeral signing key that she will be using in order to sign her messages during this session. This way she ensures her messages are authenticated. Then, every participant exchanges their ephemeral public signing keys with every other participant, using a Deniable Authenticated Key Exchange (denAKE) algorithm. When all exchanges have taken place, every participant has created $\mathcal{S}$. The ephemeral signing keys to be used in this session are deniable and hence messages signed by them are deniable as well. The DSKE sub-protocol and the denAKE are described in more detail in section 5.4.2.

During the GKA sub-protocol, the participants generate a shared secret key $\mathcal{K}$ that encryption keys will be derived from. The derived keys, in turn, will be used to encrypt the messages during this session. The GKA sub-protocol is described in more detail in section 5.4.3.

During the Attest sub-protocol, the participants authenticate sid and ensure that they agree on $\mathcal{S}$. The Attest sub-protocol is described in more detail in section 5.4.4.

During the Communication sub-protocol, the actual private conversation takes place. The users use $\mathcal{K}$, their ephemeral signing keys, and $\mathcal{S}$, in order to encrypt and authenticate their messages. When this phase is finished, a transcript of the chat room $\mathcal{T}$ is returned, which contains all the messages of the chatroom per participant. The Communication sub-protocol is described in more detail in section 5.4.5.

During the Shutdown sub-protocol, the participants determine if $\mathcal{T}$ is consistent, and reveal their ephemeral signing keys. If the transcript is indeed consistent we say that consensus has been reached. The revelation of the ephemeral signing keys adds to the deniability property of the protocol, in the same manner the key revelation in OTR protocol does. However, it's an optional feature, since the signing keys are deniable in the first place. The Shutdown sub-protocol is described in more detail in section 5.4.6.

Algorithm 7: mpOTR($\mathcal{P}$) — run a session of the mpOTR protocol

Input: $\mathcal{P}$: participants list
Result: Executes a run of the mpOTR protocol
begin
  $sid \leftarrow Offer(\mathcal{P})$
  $\mathcal{S} \leftarrow DSKE(\mathcal{P}, sid)$
  $\mathcal{K} \leftarrow GKA(\mathcal{P}, sid, \mathcal{S})$
  $\mathcal{A} \leftarrow Attest(\mathcal{P}, sid, \mathcal{S})$
  if $\mathcal{A} = \perp$ **then**
    return *"Error"*
  end
  $\mathcal{T} := Communication(\mathcal{P}, sid, \mathcal{S}, \mathcal{K})$
  $\mathcal{C} \leftarrow Shutdown(\mathcal{P}, sid, \mathcal{S}, \mathcal{T})$
  if *ConsensusForAll($\mathcal{C}$)* **then**
    return *"OK"*
  else
    return *"Error"*
  end
end

5.4 Sub-protocols

5.4.1 Offer

During the first phase of the setup procedure, which we call "Offer", the participants calculate a unique Session ID, called sid. This is a value that will be used to distinguish the current session between other sessions created by the same set of participants P.

Each participant $\hat{X}$ chooses a random 256-bit value $c_{\hat{X}}$, which is his contribution to the sid. Given an ordering rule described in 5.6, we define sid as the SHA-512 hash of the serialized ordered list that contains every participant's contribution:

$$sid = SHA512(c_{\hat{Y}_1} \| c_{\hat{Y}_2} \| \cdots)$$

The contributions are sent with an "Offer Message". An "Offer Message" sent by $\hat{X}$ contains $c_{\hat{X}}$ along with $\hat{X}$'s position in the ordered participants list according to his own perception. The position is sent for technical reasons, so that a disagreement on the participants list can be determined as early as possible.

The participant who wants to initiate the mpOTR protocol, broadcasts an "Offer Message". When other participants receive it, they also broadcast their own "Offer Message". They all store the received contributions as well as their own. Once all messages have been exchanged, all participants can calculate sid. Then, they initiate the next sub-protocol. A formal description of the Offer sub-protocol in the context of a participant $\hat{X}$ is shown in algorithm 8.

Notice that "Offer Messages" are not authenticated and hence sid must be verified *after* the participants have exchanged signing keys, as described in section 5.4.4.

Algorithm 8: Offer($\mathcal{P}$) — session ID construction in the context of participant $\hat{X}$.

Input: $\mathcal{P}$: participants list
Output: sid : the session ID
begin
  $c_{\hat{X}} \xleftarrow{\$} \{0,1\}^{256}$
  $Broadcast(c_{\hat{X}})$
  $Outstanding \leftarrow \mathcal{P} \setminus \{\hat{X}\}$
  while $Outstanding \neq \emptyset$ **do**
    $(\hat{Y}, c) \leftarrow Receive()$
    if $\hat{Y} \in Outstanding$ **then**
      $c_{\hat{Y}} \leftarrow c$
      $Outstanding \leftarrow Outstanding \setminus \{\hat{Y}\}$
    end
  end
  $sid \leftarrow SHA512(c_{\hat{Y}_1} \| c_{\hat{Y}_2} \| \cdots)$
  return sid
end

5.4.2 Deniable Signature Key Exchange (DSKE)

During the DSKE sub-protocol each participant $\hat{X}$ generates an ephemeral signing key. The public part of this key is denoted by $E_{\hat{X}}$ while the private part is denoted by $e_{\hat{X}}$. Then, they all exchange their ephemeral public signing keys in an authenticated and deniable manner. When DSKE is completed, they all have created an association table $\mathcal{S}$, which associates each participant $\hat{Y}$ to their ephemeral public signing key $E_{\hat{Y}}$.

Afterwards, the participants must make sure that they all have constructed the same $\mathcal{S}$, as described in section 5.4.4.

In [7] the following construction for a DSKE is proposed:

Given a deniable Authenticated Key Exchange (denAKE), two participants of the chat can generate a deniable and authenticated shared secret. With that secret they can exchange their ephemeral public signing key in an encrypted and authenticated fashion, using symmetric algorithms. This is done for every pair of participants. After that, they will all have created the association table $\mathcal{S}$.

denAKE

In [7] no denAKE is specified. Based on the Triple Diffie–Hellman protocol as described in section 3.10, we propose the following protocol:

1. Each participant $\hat{X}$ uses a Diffie–Hellman keypair $(A_{\hat{X}}, g^{A_{\hat{X}}})$ as a longterm key to idenifty him to other participants. This is generated only once, when a member first used the protocol, then the key remains the same for subsequent runs of the protocol.
2. Before initiating any denAKE in this session, each participant $\hat{X}$ creates an ephemeral Diffie–Hellman keypair $(a_{\hat{X}}, g^{a_{\hat{X}}})$ used only in this session. He uses the same ephemeral key to communicate with all the participants during this session.

3. Then he broadcasts the public components of the longterm and ephemeral keys to all chat participants in the tuple $(g^{A_{\hat{X}}}, g^{a_{\hat{X}}})$. We shall call this message a "Handshake Message".
4. When participant $\hat{X}$ has received a Handshake Message $(g^{A_{\hat{Y}}}, g^{a_{\hat{Y}}})$ from some other participant $\hat{Y}$, she can compute the shared secret s as specified by the Triple Diffie–Hellman protocol:

$$s = g^{a_{\hat{X}} a_{\hat{Y}}} \| g^{A_{\hat{X}} a_{\hat{Y}}} \| g^{A_{\hat{Y}} a_{\hat{X}}}$$

5. After computing s, she uses it as input of Key Derivation Functions (KDF) in order to compute AES and MAC keys.
6. Then, she encrypts-then-mac's a magic number and sends it to the other party. This is done to verify that the other party has indeed generated the same shared secret and is not an adversary trying to break the forward secrecy property of Triple Diffie–Hellman , see section 5.4.2. We shall call this message a "Confirm Message".
7. When she has received the corresponding Confirm Message, she is assured that the shared secret can be safely used and there is no foul play. Now she encrypts-then-macs her ephemeral public signing key $E_{\hat{X}}$ and sends it to the other party. This message is called a "Key Message"
8. When a key message is received, she first verifies the message using the same mac. If the tag checks out, she decrypts the key $E_{\hat{Y}}$ and adds it in the association table $\mathcal{S}$.

In figure 5.1 a schematic description of the protocol is provided.

Order of the concatenation When the shared secret is calculated, three values must be concatenated. Since concatenation is not commutative the two parties must agree on the order that the concatenation happens.

This is achieved by comparing the values $g^{A_{\hat{X}}}$ and $g^{A_{\hat{Y}}}$:

$$s = \begin{cases} g^{a_{\hat{X}} a_{\hat{Y}}} \| g^{A_{\hat{X}} a_{\hat{Y}}} \| g^{A_{\hat{Y}} a_{\hat{X}}} & g^{A_{\hat{X}}} \geq g^{A_{\hat{Y}}} \\ g^{a_{\hat{X}} a_{\hat{Y}}} \| g^{A_{\hat{Y}} a_{\hat{X}}} \| g^{A_{\hat{X}} a_{\hat{Y}}} & g^{A_{\hat{X}}} < g^{A_{\hat{Y}}} \end{cases}$$

That is, the value that is generated using the highest public key takes precedence during the concatenation. In the overview of the algorithm above, it was silently assumed that $g^{A_{\hat{X}}} \geq g^{A_{\hat{Y}}}$.

The need of a confirmation message A confirmation message is needed if we want to have forward secrecy in this exchange. We must make sure that we are really speaking with the intended participant. Consider the following scenario.

An adversary, Eve, creates an ephemeral keypair (b, g^b). Then he poses as Bob to Alice, and broadcasts a handshake message containing (g^B, g^b) where g^B is Bob's public longterm key.

After Alice receives the handshake message she can construct the shared secret. Eve however cannot construct the secret since she does not know Bob's longterm private key. If Alice starts sending data before confirming that the other party has indeed arrived at the same secret, she is under the danger to lose the forward secrecy property for all messages she sends with the secret.

Alice		**Bob**
$x \leftarrow_{\$} Z_p^*$		
	$Send\ (g^x, g^X)$ $\longrightarrow$	
		$y \leftarrow_{\$} Z_p^*$ $s \leftarrow g^{xy} \| g^{Xy} \| g^{Yx}$ $k_1 \leftarrow KDF_1(s)$ $k_2 \leftarrow KDF_2(s)$
	$Send\ (g^y, g^Y)$ $\longleftarrow$	
$s \leftarrow g^{xy} \| g^{Xy} \| g^{Yx}$ $k_1 \leftarrow KDF_1(s)$ $k_2 \leftarrow KDF_2(s)$		
	$Send$ $c = AES_{k_1}("confirm")$ $MAC_{k_2}(c)$ $\longrightarrow$	
		Verify mac $m \leftarrow AES_{k_1}^{-1}(c)$ Verify m = ”confirm”
	$Send$ $c = AES_{k_1}("confirm")$ $MAC_{k_2}(c)$ $\longleftarrow$	
Verify mac $m \leftarrow AES_{k_1}^{-1}(c)$ Verify m = ”confirm”		
	$Send$ $c = AES_{k_1}(E_{\hat{A}})$ $MAC_{k_2}(c)$ $\longrightarrow$	
		Verify mac $S[\hat{A}] \leftarrow E_{\hat{A}}$
	$Send$ $c = AES_{k_1}(E_{\hat{B}})$ $MAC_{k_2}(c)$ $\longleftarrow$	
Verify mac $S[\hat{B}] \leftarrow E_{\hat{B}}$		

Figure 5.1: The denAKE protocol, where X and Y are the private parts of the long term keys

Algorithm 9: AuthBroadcast(M) — broadcast message M authenticated under paricipant $\hat{X}$'s ephemeral signing key.

Input: M : message
$e_{\hat{X}}$: ephemeral private signing key
Result: authenticated M is brodcast
begin
$\sigma \leftarrow Sign(e_{\hat{X}}, M)$
$Broadcast(M \| \sigma)$
end

Indeed the only reason Eve can't construct the secret that Alice calculated, is that she doesn't have Bob's longterm private key. This of course means that if she somehow gets a hold of this key in the future, she will be able decrypt some messages. This situation of course does not satisfy the forward secrecy property.

Properties Triple Diffie–Hellman is a protocol that is:

1. authenticated because the shared secret can only be calculated by someone who does possess one of the longterm private keys (and the corresponding ephemeral of course);
2. forward secret because once the ephemeral key has been destroyed, it is impossible to reconstruct the shared secret even when the longterm private key is compromised;
3. deniable because the only values that are exchanged during a protocol run are the two public keys that a participant will use and nothing is signed, which means that nothing can be used to prove that someone took part in a conversation.

Another property of this protocol that comes for free is its very fast key generation as, basically, any random number can be used as a secret key.

Thus Triple Diffie–Hellman satisfies the properties required in [7] and can be used as a denAKE.

Authenticated message exchange

Once all participants have run the DSKE sub-protocol they all have constructed the association table $\mathcal{S}$ which associates each participant $\hat{X}$ to their ephemeral public signing key for this session $E_{\hat{X}}$. Each participant $\hat{X}$ also has his own ephemeral private signing key $e_{\hat{X}}$. From now on, participants can authenticate the messages exchanged in the current session.

In algorithm 9 we show how an authenticated message is sent and in algorithm 10 we show how an authenticated message is received and verified.

5.4.3 Group Key Agreement (GKA)

During the GKA sub-protocol the participants construct a shared secret key $\mathcal{K}$. The latter will be used in order for symmetric encryption keys to be derived.

The main idea is to compute a combined Diffie-Hellman-like key for all participants. To do this, each participant in the chatroom generates a private exponent x_i. Given

Algorithm 10: AuthReceive($\mathcal{S}$) — attempt to receive an authenticated message.

Input: $\mathcal{S}$: assocation table
Output: $(\hat{Y}, M)$: sender and message on success, sender and $\perp$ on failure
begin
  $(\hat{Y}, M \| \sigma) \leftarrow Receive()$
  $E_{\hat{Y}} \leftarrow \mathcal{S}[\hat{Y}]$
  if $Verify(M, \sigma, E_{\hat{Y}}) = false$ **then**
    return $(\hat{Y}, \perp)$
  end
  return $(\hat{Y}, M)$
end

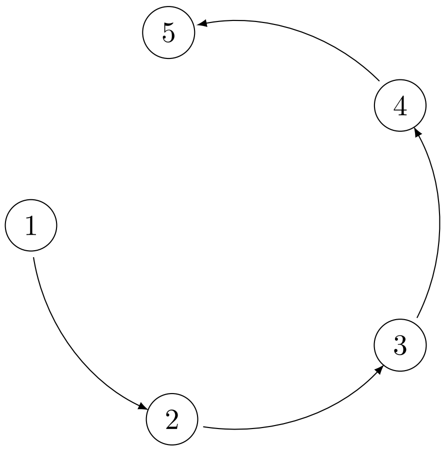


FIGURE 5.2: This diagram demonstrates the upflow of the intermediate keys

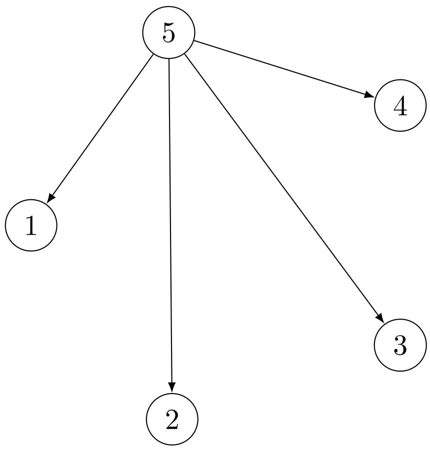


FIGURE 5.3: This diagram demonstrates the downflow of the intermediate keys

that g is the shared base and n the number of participants, the shared secret $\mathcal{K}$ will be:

$$\mathcal{K} = g^{x_1 \cdot x_2 \cdots x_n}$$

The key agreement is performed in n steps. Given the ordering rule of participants described in 5.6, in each step a participant receives an intermediate key list from the previous participant, calculates a new intermediate key list based on the received one and sends the new list to the next participant.

In the first step, no previous list is received and the new list is created from scratch. In the last step, the new list is broadcasted to all other participants. We call each of the $n-1$ first steps "Upflow", and the last step "Downflow".

Upflows

The first participant sends a list containing g and g^{x_1} where x_1 is his private exponent. To construct the new key list, each next participant prepends a copy of the last element to the reveived list, and raises every other element to his private exponent. These messages shall be called "Upflow Messages" and the Upflow forwarding procedure is illustrated in figure 5.2

Algorithm 11: SendUpflow($InterKeys$, x, $\hat{Y}$) — send the new intermediate key list to the next participant.

Input: $InterKeys$: previous intermediate key list
x : user's secret key
$\hat{Y}$: the next participant
Result: Sends the new intermediate key list to the next participant

```
begin
    inter_key_list ← []
    inter_key_list.Append(InterKeys.Last())
    foreach k in InterKeys do
        inter_key_list.Append(k^x)
    end
    AuthBroadcast(Ŷ‖inter_key_list)
end
```

Algorithm 12: SendDownflow($InterKeys$, x) — broadcast the downflow intermediate key list to the other participants.

Input: $InterKeys$: previous intermediate key list
x : user's secret key
Result: Broadcasts the downflow intermediate key list to the other participants

```
begin
    inter_key_list ← []
    foreach k in InterKeys do
        inter_key_list.Append(k^x)
    end
    AuthBroadcast(inter_key_list)
end
```

Downflow

Once the last participant has received the Upflow, he calculates $\mathcal{K}$ by raising the last element of the received list to his private exponent. Then, he constructs the final intermediate key list by removing that last element and raising all other elements of the received list to his private exponent. He finally sends the final key list back to all previous participants, as illustrated in figure 5.3. This message, containing the final intermediate key list shall be called "Downflow Message".

All other participants now can also caclulate the shared secret $\mathcal{K}$. The $i-th$ participant calculates $\mathcal{K}$ by raising $i-th$, **counting from the end**, element of the final intermediate key list to his private exponent. Then, they can use $\mathcal{K}$ for encryption.

Example

To illustrate the intermediate key list construction we give an example of a GKA between 5 participants with private exponents a, b, c, d and e, so that:

$$\mathcal{K} = g^{abcde}$$

The messages exchanged contain the following intermediate key lists:

Algorithm 13: GKA($\mathcal{P}$, *sid*, $\mathcal{S}$) - execute a Group Key Agreement and produce the shared secret in the context of participant $\hat{X}$.

```
Input: 𝒫 : participants list
       sid : the session ID
       𝒮 : association table
Output: 𝒦 : the shared secret
begin
    x ← GenerateKey()
    Ŷ_prev ← X̂.Previous()
    Ŷ_next ← X̂.Next()
    if Ŷ_prev = NULL then                                  /* X̂ is first */
        SendUpflow([G], x, Ŷ_next)
    else
        repeat                              /* wait for previous upflow */
            (Ŷ, R̂‖key_list) ← AuthReceive(𝒮)
        until R̂ = X̂;
        if Ŷ ≠ Ŷ_prev ∨ R̂‖key_list =⊥ then
            return error
        end
        if Ŷ_next ≠ NULL then                /* X̂ is not first or last */
            SendUpflow(key_list, x, Ŷ_next)
        else                                                /* X̂ is last */
            final_key ← key_list.Last()
            𝒦 ← final_key^x
            SendDownflow(key_list, x)
            return 𝒦
        end
    end
    repeat                                          /* wait for downflow */
        (Ŷ, key_list) ← AuthReceive(𝒮)
    until Ŷ = 𝒫.Last();
    if key_list =⊥ then
        return error
    end
    pos ← 𝒫.IndexOf(X̂)
    final_key ← key_list.Reverse().Get(pos)
    𝒦 ← final_key^x
    return 𝒦
end
```

$$1 \rightarrow 2 : g, g^a$$
$$2 \rightarrow 3 : g^a, g^b, g^{ab}$$
$$3 \rightarrow 4 : g^{ab}, g^{ac}, g^{bc}, g^{abc}$$
$$4 \rightarrow 5 : g^{abc}, g^{abd}, g^{acd}, g^{bcd}, g^{abcd}$$
$$5 \rightarrow all : g^{abce}, g^{abde}, g^{acde}, g^{bcde}$$

It's obvious how participant 5 can calculate $\mathcal{K}$ using the last element of the list received, g^{abcd}, and his private exponent e. It's also obvious how each of the rest participants can calculate $\mathcal{K}$ using his own private exponent and the proper element of the final list.

Detailed description

For a more detailed description of the sub-protocol we refer to [9], however a pseudocode is provided.

Algorithm 11 presents an overview of how each upflow message is constructed, using the data received from the previous upflow message.

Algorithm 12 presents an overview of how the final, downflow message is constructed using the data received from the last upflow message.

In algorithm 13 the two previous algorithms (11 and 12) are used in order to execute a complete run of the GKA protocol.

5.4.4 Attest

During this sub-protocol the participants must verify that they agree on the sid and the association table $\mathcal{S}$. This is needed for two reasons. First, because sid is required before $\mathcal{S}$ is constructed, and therefore the messages exchanged for the calculation of sid can't be signed. Second, because the participants need to verify that they all have the same view of $\mathcal{S}$.

Each participant calculates the SHA-512 hash of $\mathcal{S}$, which is sorted according to the rule described in 5.6. Then they broadcast an authenticated message, called "Attest Message", which contains both the sid and the calculated hash. To authenticate the message, each sender signs it using the ephemeral private signing key $e_{\hat{X}}$ corresponding to the ephemeral public signing key $E_{\hat{X}}$ that has now exchanged with the other participants.

When a participant receives an "Attest Message", first she must verify the signature. Then she must check for two things. First, verify the received hash of $\mathcal{S}$ matches the one she computed herself. Second, check that the received message uses the expected sid. A formal description of the Attest sub-protocol in the context of a participant $\hat{X}$ is shown in algorithm 14.

Provided that SHA-512 is a cryptographic hash function an attacker can't find two signing keys association tables with the same hash. This means that he is not able to make two participants believe that they have arrived at the same table when they have not. Also by signing the message containing the specific sid, a user implicitly verifies that he is using that particular sid for this session.

Algorithm 14: Attest($\mathcal{P}$, sid, $\mathcal{S}$) — authenticate previously unauthenticated protocol parameters for the current session in the context of participant $\hat{X}$.

Input: $\mathcal{P}$: participants list
sid : the session ID
$\mathcal{S}$: the association table
Output: $\mathcal{A}$: "OK" if verification went good, $\perp$ if it went wrong

```
begin
    M ← sid‖SHA512(S)
    AuthBroadcast(M)
    Outstanding ← P\{X̂}
    while Outstanding ≠ ∅ do
        (Ŷ, M_Ŷ) ← AuthReceive(S)
        if M_Ŷ = ⊥ ∨ M_Ŷ ≠ M then
            return ⊥
        else
            Outstanding ← Outstanding\{Ŷ}
        end
    end
    return "OK"
end
```

5.4.5 Communication

Using the Communication sub-protocol, the participants can exchange authenticated and encrypted messages using the association table $\mathcal{S}$ derived from DSKE and the shared key $\mathcal{K}$ derived from GKA.

Origin authentication

For origin Authentication we use public key encryption methods. This is done because use of symmetric algorithms would require a participant who wishes to send a message to mac the message $n-1$ times, where n is the number of the participants.

Algorithm While describing the DSKE we mentioned that an ephemeral signing key is transmitted by each participant to every other, to be used for message origin verification.

For this purpose we make use of the EdDSA algorithm. This algorithm was selected for its fast key generation, since a new keypair must be generated in each protocol run, and its relatively small signature size.

Signing The signature generation is the last step taken before sending a message. This way we can sign all the properties of the message to be sent, like the sid or the recipient (if any). We also avoid any manifestations of the Cryptographic Doom Principle, which states that if a protocol tries to perform *any* cryptographic operation before verifying the signature or mac on a received message, it will somehow fail catastrophically and lead to doom.

Symmetrically the signature verification is the first thing that happens before any other operation is performed on the received message (cryptographic or not).

Encryption

For encryption a shared secret key $\mathcal{K}$ is used by all the members. This is not a problem since the origin authentication is provided by the signatures, and we obviously don't mind any chat member to read a message or we wouldn't participate in the chat in the first place.

To encrypt a message, a user concatenates the shared secret with his personal id and creates a personal key. This is the actual encryption key.

$$k_{enc} = H(id_{personal} \| master\ key)$$

For the counter, each user stores locally his personal upper half (8 most significant bytes). The lower half (8 least significant bytes) are always set to zero. The top half of the counter is prepended in the sent message. The ciphertext is produced as follows (where ctr is the top half of the locally stored counter):

$$ciphertext = AES_{CTR}(k_{enc}, ctr\|0, plaintext)$$

To decrypt a message, a user concatenates the shared secret with the id of the message's sender.

$$k_{dec} = H(id_{sender} \| master\ key)$$

He uses the prepended top half of the counter.

$$plaintext = AES_{CTR}(k_{dec}, ctr\|0, ciphertext)$$

This encryption scheme is used, so that the possibility that a certain encryption key and counter pair is eliminated. If all the participants used the shared secret itself as an encryption key and two users sent a message at the same time, they would use the same counter. This would be a catastrophic failure. We discuss why we choose Counter mode in section 5.5

Transcript

In order for the shutdown sub-protocol to be executed we need to store the transcript of the chatroom. In reality a seperate transcript is held for the messages from each participant. The shutdown protocol will then combine all the different transcripts to determine if consensus has been reached.

The transcripts are implemented as linked lists. Each list is kept sorted in lexicographic order. When a message is to be added in a transcript list, the list is searched linearly to find the position the new message should be placed at.

When user A sends a message, he adds that message to the transcript corresponding to himself. When he receives a message from user B, he adds that message to the transcript corresponding to user B.

5.4.6 Shutdown

During the Shutdown sub-protocol, the participants end the current session. Chat room transcript consistency is checked and the ephemeral signing keys are published, in order to permit a posteriori modifications of the chat transcript. The revelation of the ephemeral signing keys adds to the protocol's deniability property. However,

Algorithm 15: Shutdown($\mathcal{P}$, sid, $\mathcal{S}$), $\mathcal{T}$) — called in the context of participant $\hat{X}$, determines if consensus has been reached with other participants and publishes the ephemeral signing key of $\hat{X}$.

Input: $\mathcal{P}$: participants list
sid : the session ID
$\mathcal{S}$: association table
$\mathcal{T}$: chat room table of transcripts
$e_{\hat{X}}$: ephemeral private signing key

Output: $\mathcal{C}$: boolean array of consensus status for each participant, $\perp$ if shutdown went wrong

begin
- `// Broadcast digest of our transcript`
- $h_{\hat{X}} \leftarrow SHA512(\mathcal{T}[\hat{X}])$
- $AuthBroadcast(\text{"}shutdown\text{"} \| h_{\hat{X}})$
- `// Collect digests of others' transcripts`
- $Outstanding \leftarrow \mathcal{P} \backslash \{\hat{X}\}$
- **while** $Outstanding \neq \emptyset$ **do**
  - $(\hat{Y}, M \| h'_{\hat{Y}}) \leftarrow AuthReceive(\mathcal{S})$
  - **if** $\hat{Y} \in Outstanding \wedge M = \text{"}shutdown\text{"}$ **then**
    - $h_{\hat{Y}} \leftarrow SHA512(\mathcal{T}[\hat{Y}])$
    - $Outstanding \leftarrow Outstanding \backslash \{\hat{Y}\}$
  - **end**
- **end**
- `// Broadcast digest of full chat`
- $h \leftarrow SHA512(h_{\hat{Y}_1} \| h_{\hat{Y}_2} \| \cdots)$
- $AuthBroadcast(\text{"}digest\text{"} \| h)$
- `// Determine consenus`
- $Outstanding \leftarrow \mathcal{P} \backslash \{\hat{X}\}$
- **while** $Outstanding \neq \emptyset$ **do**
  - $(\hat{Y}, M \| h') \leftarrow AuthReceive(\mathcal{S})$
  - **if** $\hat{Y} \in Outstanding \wedge M = \text{"}digest\text{"}$ **then**
    - $consensus[\hat{Y}] \leftarrow h = h'$
    - $Outstanding \leftarrow Outstanding \backslash \{\hat{Y}\}$
  - **end**
- **end**
- `// Assure that everybody aborted the session`
- $AuthBroadcast(\text{"}end\text{"})$
- $Outstanding \leftarrow \mathcal{P} \backslash \{\hat{X}\}$
- **while** $Outstanding \neq \emptyset$ **do**
  - $(\hat{Y}, M) \leftarrow AuthReceive(\mathcal{S})$
  - **if** $M \neq \text{"}end\text{"}$ **then**
    - **return** $\perp$
  - **else**
    - $Outstanding \leftarrow Outstanding \backslash \{\hat{Y}\}$
  - **end**
- **end**
- `// Reveal the ephemeral private signing key`
- $Broadcast(e_{\hat{X}})$
- **return** $consensus$

end

the protocol is deniable even without publishing the ephemeral signing keys, since the signing keys are deniable in the first place.

For a session to be terminated, a ”Shutdown Message” is sent. This message signals to other participants that the shutdown phase should be initiated. It contains the hash of all the messages sent by the user sending the ”Shutdown Message”. When a user receives a ”Shutdown Message”, he also responds with a ”Shutdown Message” containing his own messages hash, if he hasn't already sent one.

When one participant has received a ”Shutdown Message” from all other participants he can send a ”Digest Message”. This message contains a digest of all the messages in the chat room and is calculated as follows:

> Sort the participants using their usernames in lexicographic order.
>
> For each participant i (in that order) calculate the hash $h_i = H(S_i)$, where S_i is set of all the messages sent by this user, sorted in lexicographic order too.
>
> Calculate the digest $h = H(h_1 \| h_2 \dots \| h_N)$, where N is the number of participants.

When one participant receives a ”Digest Message” from some other participant, he checks whether the two of them agree on the chat room transcript. He simply compares the digest he computed locally to the one sent by the other participant. He deduces that consensus is reached only if the two digests are the same.

When one participant has received a ”Digest Message” from every other participant, he broadcasts an ”End Message”. This message signifies that the sender will not use the channel to send any messages anymore.

When a participant receives an ”End Message” from all other participants, he is certain that he can release his signing secret key. This is done by broadcasting a message which we shall call ”Key Release Message”. Now anyone who intercepts the released key can forge chat room messages. However all the participants will no longer accept messages signed with the released secret key, and thus it cannot be used to impersonate its previous owner.

A formal description of the Shutdown sub-protocol in the context of a participant $\hat{X}$ is shown in algorithm 15.

5.5 The primitives

In the protocol description we talked generally about Diffie–Hellman Key exchanges, encryption ciphers and signatures. Here we shall present the specific algorithms and parameters we used in our implementation.

5.5.1 Diffie–Hellman Group

The already existing libotr implementation of the Diffie–Hellman key exchange is used. As a result we use classic Diffie–Hellman and specifically the group no. 5 [8] with a 1536 bit modulus. In the algorithms presented above, all exponentiations are performed in this group.

5.5.2 Encryption

For encryption we use AES-128 in CTR mode, the same cipher as the two-party OTR protocol.

We chose 128-bit AES instead of the 256-bit one because our Diffie–Hellman group does not provide 256 bit of entropy and evidence suggests that the 128-bit AES key schedule is preferred for security [1] [4].

We chose CTR mode because it is malleable. This increases the deniability properties of the protocol, as described in [11] and [2].

5.5.3 Authentication

For signing we use the EdDSA algorithm over the Ed25519 curve. This signature scheme was chosen primarily because of its fast key generation. Each message is signed as a whole. This means that the signature covers the message and any metadata sent, like session id, counter value etc.

5.6 Participants ordering

In many cases, the protocol demands some ordering of the participants. For example, in the Offer step of the Setup phase, the sid contributions should be concatenated in some order before hashing them. Same stands for the public signing keys in the association table, during the Attest step. So we define an ordering rule for the participants, that is used whenever an ordering of elements corresponding to participants is needed.

Given that the application provides the mpOTR protocol with a unique name describing each participant, we make the convention that every ordering of the participants list is made lexicographically based on that unique name. We also define the position of the participant, as his position in this order, starting counting from zero (0).

5.7 Identity verification

The identity of a participant is verified during the DSKE phase. In order for a participant to be verified the fingerprint of his longterm key must be stored in the known fingerprints file, and be explicitly marked as verified. In our implementation this file is comma separated and is of the form:

```
<account_name>,<protocol>,<buddy_name>,<fingerprint>,<is_verified>
```

The account name is the "address" of the library user in the form username@host, and is provided by the application. The protocol is a string that is characteristic of the underlying protocol that the specific account uses. It is again provided by the application. The buddy name is the nickname that a user is identified with. For the time being it is the username part of a users "address". This implies that the underlying protocol provides addresses which do not change often for a user. It also means that users from multiple hosts are not allowed or else two users with the same username might conflict. The above holds for a protocol like jabber for example, but is not the case for protocols like IRC. As a result our implementation is not fully protocol agnostic at this moment. Finally the fingerprint is the SHA-256 hash of the public identity key, and the last field is 1 if the key is verified and 0 otherwise.

This file is read during the plugin start up and initializes the list of all known fingerprints. When a new chatroom is created, each participant is assigned a list with all known fingerprints used by this participant (from the above list). When a "Handshake Message" is received the participants known fingerprints list is checked to find (if it exists) a fingerprint matching the currently used public key. If such a fingerprint is found and it is verified, then the user is verified for this session. If the found fingerprint is not verified then the participant is not verified. If no key is found then again the participant is not verified and an entry for this new key is added in the known fingerprints list.

5.8 Handling out-of-order messages

In the protocol description above, we silently assumed that messages are delivered in-order. In fact, in a distributed context, like the mpOTR, this can be really tricky.

Specificaly in our construction, there are quite a lot messages exchanged during the first four sub-protocols that run automatically to setup the session parameters and hence these messages are usually exchanged in a narrow period of time. Latency differences between participants, either due to the underlying network or due to the participants' equipment may result in out-of-order message delivery to some participants[1].

For example, there exists a possible scenario where all Offer Messages have been sent, but some of them have only reached some participants in some point of time. In that case, those who have received the Offer Messages proceed to the DSKE sub-protocol and send their Handshake Messages. The latter, may reach participants that haven't yet finished the Offer sub-protocol, and obviously cannot be handled by them.

So, there is a need for some mechanism that ensures that mpOTR messages are handled in-order by the mpOTR Protocol. There are two general approaches to solve this problem in a distributed context.

The first possible approach is the utilization of some synchronization mechanism between the participants so that each participant would ensure that all other participants can handle a message before he sends it. However, this is a rather complicated approach because it would require more messages to be exchanged so that the participants could signal points of synchronization amongst themselves.

We utilized the second approach, which is to make the receivers responsible for handling the received messages in-order. This is a rather simple approach, since it only requires each participant to keep out-of-order messages in a pending queue and handle them when messages supposed to be received ealrier have indeed been received and handled.

5.9 Message Structure

5.9.1 Top level encoding

We follow the OTR rules for the top level encoding of mpOTR messages. That is, all mpOTR messages start with the string "?OTR:" followed by the base64 encoded message and then a "." denoting the end of the message. For a mpOTR message M, we send:

[1] In fact, this was one of the first complications we faced during our implementation.

$$"?OTR:"\|Base64(M)\|"."$$

5.9.2 Instance tags

Many IM Protocols allow for a client to be logged into the same account from multiple locations and the server may relay messages to all of them. The version 3 of OTR utilizes a distinguishing mechanism, called instance tags.

Instance tags are 32-bit values that are intended to be persistent. If the same client is logged into the same account from multiple locations, the intention is that the client will have different instance tags at each location. All mpOTR messages include the sender's instance tag.

5.9.3 Data Types

We present a list of several data types used in mpOTR messages along with their encoding:

SHORT Short integers. 2 byte unsigned value, big-endian.

INT Integers. 4 byte unsigned value, big-endian.

MPI Multi Precision Integers. 4 byte unsigned len, big-endian. len byte string with each byte of the MPI encoded as 2 hex digits. Negative numbers are prefix with a minus sign and in addition the high bit is always zero to make clear that an explicit sign ist used.

LIST Lists of any other type. 4 byte unsigned len, big-endian. len encoded list elements.

HASH32 Hashes. 32 bytes of data.

HASH64 Hashes. 64 bytes of data.

CTR AES CTR-mode counter value. 8 bytes data.

DATA Opaque variable-length data. 4 byte unsigned len, big-endian. len byte data.

5.9.4 Message types

There are 12 different types of mpOTR messages. Each type is indicated by a positive integer. We have reserved the zero value to indicate non-OTR messages. The list of these types along with the corresponding integer value is shown below:

0 Non-OTR Message

1 Offer Message

2 Handshake Message

3 Confirm Message

4 Key Message

5 Upflow Message

6 Downflow Message

7 Attest Message

8 Data Message

9 Shutdown Message

10 Digest Message

11 End Message

12 Key Release Message

5.9.5 General message structure

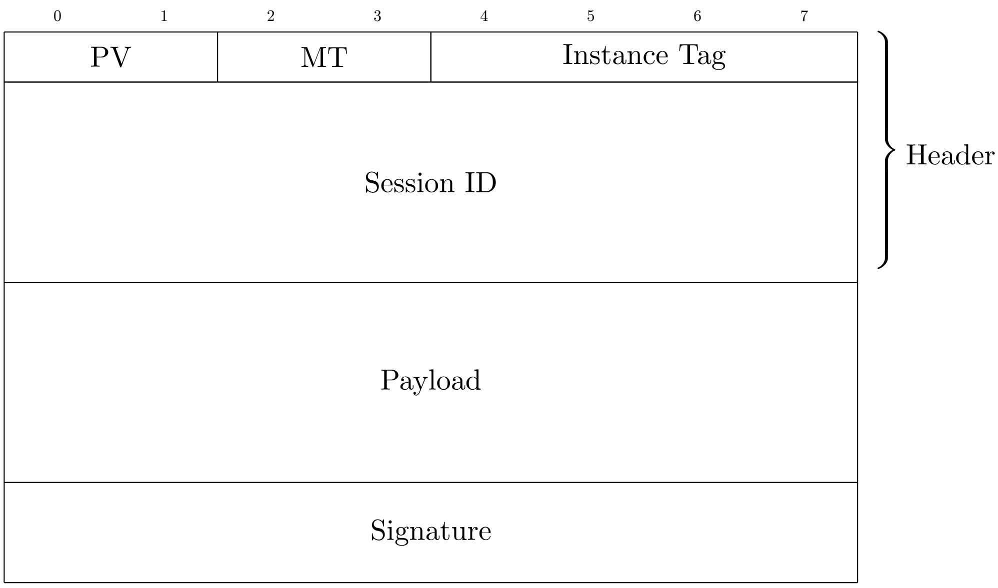


FIGURE 5.4: General message structure

In figure 5.4 the general structure of mpOTR messages is shown. Each message contains the following fields:

- PV: the Protocol Version. A SHORT indicating the Protocol Version the message is intended for.
- MT: the Message Type. A SHORT indicating the type of the message. See 5.9.4 for more details.
- Instance Tag: an INT indicating the instance tag. See 5.9.2 for more details.
- Session ID: a HASH64 containing the value of sid. Not existent in Offer Messages.
- Payload: specific data dependent for each type of message. See 5.9.6 for more details.
- Signature: the signature covering all data above, used to authenticate the message. Not existent in all message types.

There are only a few exceptions. Offer Messages contain no Session ID in the header, since sid has not been established yet. Offer and DAKE Messages contain no Signature since the association table $\mathcal{S}$ has not been constructed yet. Shutdown KeyRelease Messages also contain no Signature since authentication is not required.

5.9.6 Payload structures

In the following paragraphs we show the payload structure of each mpOTR message type. We also briefly describe each field.

Offer Message Payload

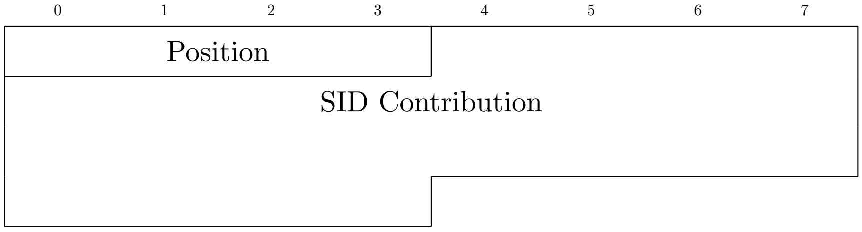


FIGURE 5.5: The structure of Offer Message payload

- Position: an INT indicating the sender's position in the ordered list of the participants.
- SID Contribution: a HASH32 containing the sender's contribution for the construction of the sid.

Handshake Message Payload

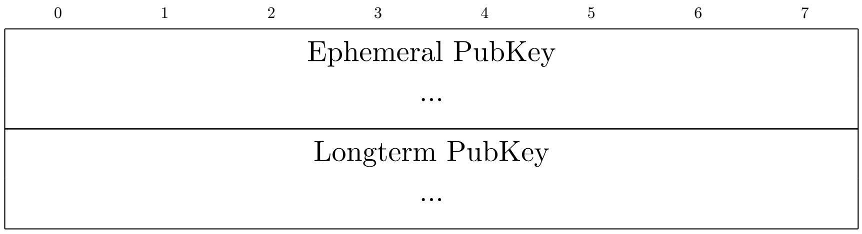


FIGURE 5.6: The structure of Handshake Message payload

- Ephemeral PubKey: a MPI containing the public part of the ephemeral key.
- Longterm PubKey: a MPI containing the public part of the longterm key.

Confirm Message Payload

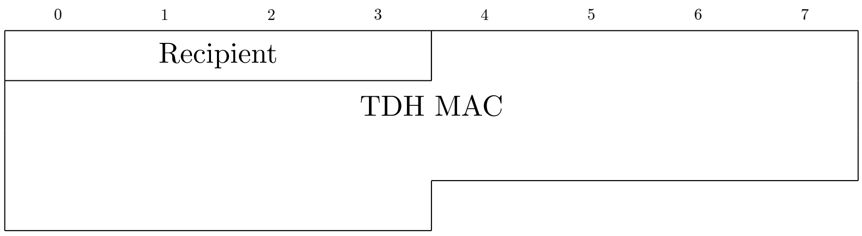


FIGURE 5.7: The structure of Confirm Message payload

- Recipient: an INT indicating the participant this message is intended for. It's actually his position in the ordered participants list.
- TDH MAC: a Triple Diffie-Hellman MAC. A HASH32 containing the MAC value for a magic number.

Key Message Payload

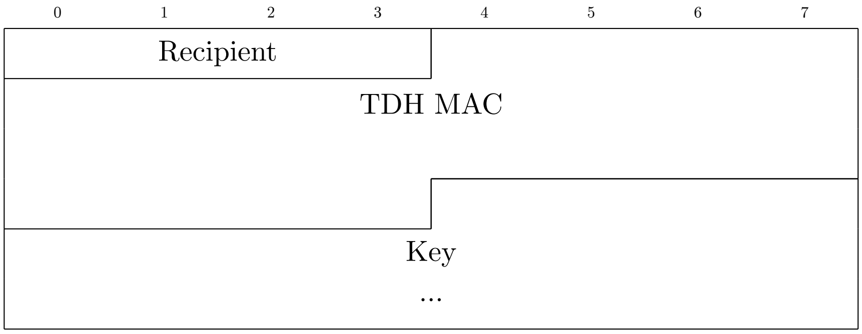


FIGURE 5.8: The structure of Key Message payload

- Recipient: an INT indicating the participant this message is intended for. It's actually his position in the ordered participants list.
- TDH MAC: a Triple Diffie-Hellman MAC. A HASH32 containing the MAC value for the following key.
- Key: DATA containing the encrypted public part of the ephemeral signing key.

Upflow Message Payload

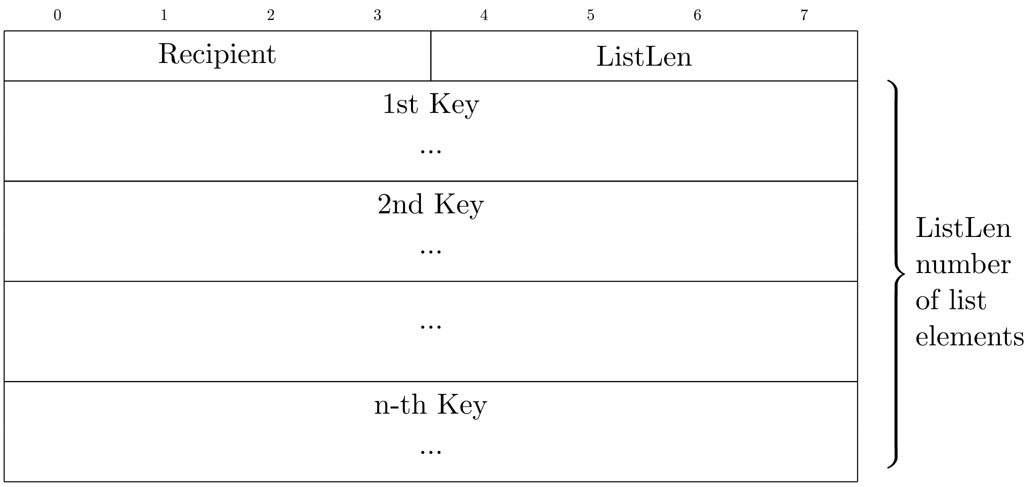


FIGURE 5.9: The structure of Upflow Message payload

- Recipient: an INT indicating the participant this message is intended for. It's actually his position in the ordered participants list.
- ListLen: an INT containing the length of the following key list.
- i-th Key: an MPI containing the i-th key of the intermediate key list.

Downflow Message Payload

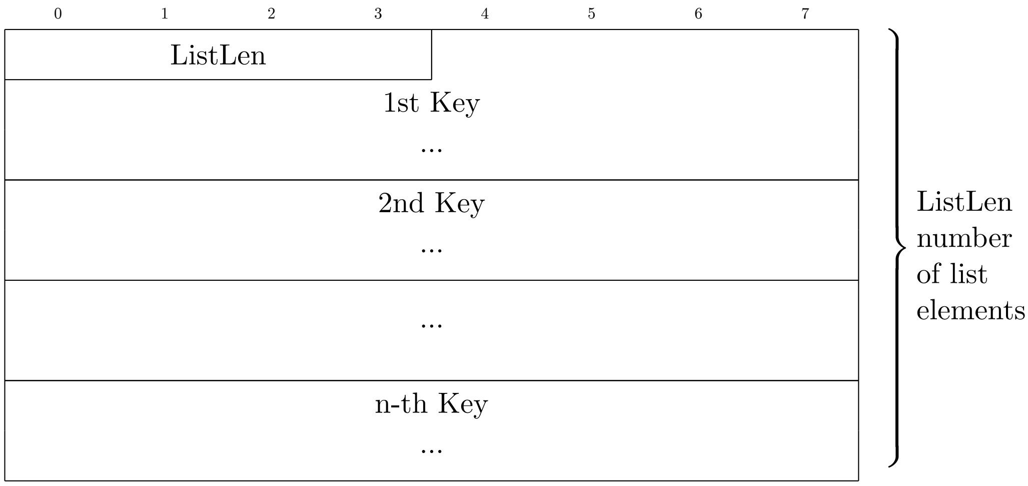


FIGURE 5.10: The structure of Downflow Message payload

- ListLen: an INT containing the length of the following key list.
- i-th Key: an MPI containing the i-th key of the intermediate key list.

Attest Message Payload

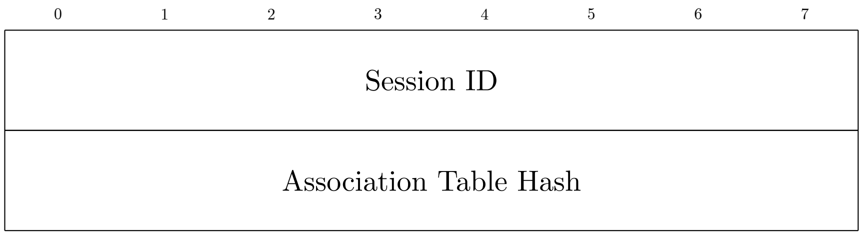


FIGURE 5.11: The structure of Attest Message payload

- Session ID: a HASH64 containing the value of sid.
- Association Table Hash: a HASH64 containing the hash of the association table $\mathcal{S}$.

Data Message Payload

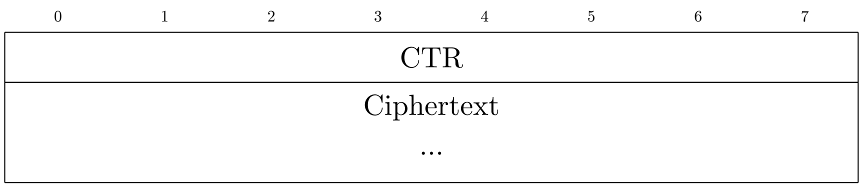


FIGURE 5.12: The structure of Data Message payload

- CTR: a CTR containing the top half of the counter for AES CTR-mode.
- Ciphertext: a DATA containing the encrypted message.

Shutdown Message Payload

<table><tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr><tr><td colspan="8">Shutdown Hash</td></tr></table>

FIGURE 5.13: The structure of Shutdown Message payload

- Shutdown Hash: a HASH64 containing the hash of the sender's transcript.

Digest Message Payload

<table><tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr><tr><td colspan="8">Digest</td></tr></table>

FIGURE 5.14: The structure of Digest Message payload

- Digest: a HASH64 containing the hash of all transcripts.

End Message Payload

End Messages contain no payload.

Key Release Message Payload

<table><tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr><tr><td colspan="8">Key
...</td></tr></table>

FIGURE 5.15: The structure of Key Release Message payload

- Key: a DATA containing the value of the key to be revealed.

Chapter 6

Implementation

6.1 Summary

Our primary goal was to design and implement the protocol we specified as a production-grade software, aiming to meet the needs of a wide user base. Of course every user would expect of such a software to offer privacy in communication between two parties, too. That said, implementing mpOTR as part of the OTR library was a natural decision. Not only would this result in a complete IM privacy library, but it would, as well, apply to an already existing wide user base.

Quite a few implementations of the OTR Library exist, but only two of them have been actually developed by the OTR Development Team. The first one is implemented in C, it's the very first implementation and the most actively developed having 4 major versions with latest release in March of 2016. The other one is implemented in Java, and has only one release in October of 2009. We chose to develop mpOTR as part of the C implementation of the OTR Library.

6.2 Designing the Integration

Integrating a new feature into an existing software is quite a challenge. Ideally, a good design would at least follow the same coding style, make the best possible reuse of the existing code and follow the same design patters. However, after a carefull inspection of the OTR Library source code we realized that following this approach was unfeasible.

First of all, the coding style in OTR Library source code is inconsistent. Different characters have been used for indentation, there is no standard error handling style, etc. Reusing parts of the existing code was not an option most of the time due to extensive coupling between the various modules. Finally, no specific design patterns had been used in the existing code.

Our approach was rather different. First, we used the coding style used more frequently in the existing code. The code reuse was limitted to the Diffie–Hellman and Base64 implementations that were the only reusable components of the existing code. As for the design patterns, we decided to use them based on theory.

6.3 Design Challenges in C

A great deal of the callenges a software engineer is going to face when designing a software to be developed in C origins in the lack of literature regarding the Design Patterns. Most of the relative literature, such as the commonly referenced [5], describe the actual implementations of the patterns in the context of an object oriented design.

Given that C is not an object oriented language, a developer should be innovative when implementing commonly used design patterns. Fortunately, C is a powerfull language offering the mechanisms to implement almost any design pattern. This power mainly comes from two features, the ability to specify incomplete types in order to achieve abstraction in the sense of information hiding, and the use of *void** in order to achieve generality as interface and inheritence would do in an object oriented context. The latter must be used carefully, since it could raise type-safety risks.

The most complete reference of design patterns in C can be found in [13]. Although it only covers a small number of patterns, it equips the reader with a clear approach of designing patterns when object-oriented techniques are not natively supported by the language. We used [13] as a reference for various patterns we implemented.

6.4 Design Patterns

In this section we describe the Design Patterns we utilized in our implementation.

6.4.1 First-Class ADT

First-Class ADT is a pattern that decouples interface from implementation, thus improving encaplulation and providing loose dependencies. We get a definition from [13]:

> ADT stands for Abstract Data Type and it is basically a set of values and operations on these values. The ADT is considered first-class if we can have many, unique instances of it.

Our implementation of First-Class ADTs is based on the paradigm found in [13].

The header file of each First-Class ADT contains the declaration of a pointer to an incomplete type and the declaration of all functions that the interface constists of. The source file of each First-Class ADT contains the definition of the incomplete type as a structure and the definition of each interface function.

Instances of the declared pointer serve as a handle for the clients. This mechanism enforces the clients to use the provided interface functions rather than directly accessing the fields of the structure.

We implemented most of the infrastructure components as First-Class ADTs, as described in section 6.6.3.

6.4.2 Observer

In [13] a C implementation of the Observer Pattern is proposed. It's a complete approach enabling an arbitrary number of observers to attach to an event emitter.

In our mpOTR implementation, the library utilizes an event emitting mechanism meant to signal different types of events to the client application. In our case, only one observer exists. So, our implementation of the pattern is simpler. The client-application (observer), attaches to the mpOTR library (event emitter) by providing a pointer to its event handling function.

6.4.3 Iterator

Our implementation utilizes lists of several entities. We implemented the list as a First-Class ADT, so that the the internal structure remains hidden from higher level components. In order for the list client to traverse the list and access the list's elements we had to follow the Iterator Pattern. We implemented the list iterator as a First-Class ADT, too.

6.5 Idiomatic Expressions

Idiomatic expressions, also reffered to as idioms, are low-level patterns that depend upon the implementation technology and usually are only applicable in a specific language. We briefly explain the ones we used, that otherwise could seem strange when reading the source code.

6.5.1 Constants to the left

The C language allows assignments inside conditional statement. This can be convenient in various cases, but can also lead to serious bugs when a programmer accidentaly uses the assignment symbol instead of the comparison symbol, as in the following example:

```
if(x = 0) {
    /* This will never be true */
}
```

By keeping constants to the left in comparisons the compiler will catch an erroneous assignment. So, a correct conditional statement using the idiom looks like:

```
if(0 == x) {
    /* Control will get here if x is zero */
}
```

6.5.2 Sizeof to variables

When using functions for dynamic memory allocation in the C language the client has to provide the required size information. Many programmers use to dynamically allocate memory as in the following example:

```
OneType* var = malloc(sizeof(OneType));
```

Code like this contains a subtle form of dependency, that is the size given as an argument of malloc must match the size of the left sife of the assignment. A change in the variable type requires a change in two places. A failure to update both places would leave the code with undefined behaviour.

By applying *sizeof* to variables the dependency is removed and the change is limited to one place. When the idiom is applied the code will look like this:

```
OneType* var = malloc(sizeof *var);
```

6.6 Design Architecture

In this section we present the different components that our mpOTR Library implementation consists of. We follow a top-down approach starting from the top-level component and then presenting lower-level components.

6.6.1 Top-level protocol component

The top-level component of our implementation is called *chat_protocol*. It incorporates the basic API endpoints that start a private session, handle received or about to send messages and end a private session. It also provides API endpoints for private identity key management and known fingeprints management, that are actually wrappers of functions provided by other components. For a detailed API description see section 6.7.

A great portion of the mpOTR Protocol implementation is carried out when handling received messages, since this is when state transitions occur. Regarding the received messages, the protocol component is responsible to check if session ID matches, verify the signature of signed messages and finally sort them out forwarding them to the proper sub-protocol component, adding them to the pending queue or discarding them. After a message is forwarded to a sub-protocol component, the protocol component checks if a state transition occured, and initializes the subsequent subprotocol as needed.

Regarding the messages that are about to be sent, it's responsible to check if a private session has been set up. In that case, it should sign and encrypt them properly or discard them if any error occurs.

The code of the protocol component is listed in appendix A′.

6.6.2 Sub-protocol components

Sub-protocols are implemented as separate components. The general form of each one incorporates an implementation of a First-Class ADT holding internal sub-protocol information. It also provides an interface to be utilized by the top-level protocol component.

For each chat room, the corresponding *ChatContextPtr* instance holds a handle of the incorporated First-Class ADT for each sub-protocol. Received messages are forwarded by the top-level protocol to the proper sub-protocol along with the *ChatContextPtr* instance handle. *ChatContextPtr* is described in section 6.6.3.

Each sub-protocol impplements its own state machine. The current state is provided by the interface of the First-Class ADT so that state transitions can be determined by the top-level protocol component.

The First-Class ADT handle type is named *Chat[...]InfoPtr* and the state type *Chat[...]State*, where *[...]* is the name of the subprotocol.

The basic interface, common in most of sub-protocols, contains the following functions:

```
void chat_[...]_info_free(Chat[...]InfoPtr info);
Chat[...]State chat_[...]_info_get_state(Chat[...]InfoPtr info);
int chat_[...]_init(ChatContextPtr ctx, ChatMessage **msgToSend);
int chat_[...]_is_my_message(const ChatMessage *msg);
int chat_[...]_handle_message(ChatContextPtr ctx, ChatMessage
    *msg, ChatMessage **msgToSend);
```

Listing 6.1: General sub-protocol interface

chat_[...]_info_free: Frees the internal data of the First-Class ADT specified by *info* handle.

chat_[...]_info_get_state: Returns the current state of the sub-protocol specified by the *info* handle.

chat_[...]_init: Initiates the sub-protocol regarding the chat room indicated by *ctx*. Output parameter **msgToSend* returns a message that should be sent after the initialization, if any. Returns *0* if no error occured, *non-zero* in case of an error.

chat_[...]_is_my_message: Returns *1* if the message should be handled by this sub-protocol or *0* if not.

chat_[...]_handle_message: Handles the message *msg* received in the chat room specified by *ctx*. Output parameter **msgToSend* returns a message that should be sent after handling, if any. Returns *0* if no error occured, *non-zero* in case of an error.

Offer

The states of the offer sub-protocol are defined as following:

```
typedef enum {
    CHAT_OFFERSTATE_NONE,
    CHAT_OFFERSTATE_AWAITING,
    CHAT_OFFERSTATE_FINISHED
} ChatOfferState;
```

Listing 6.2: Offer states

In addition to the general sub-protocol interface shown in listing 6.1, offer interface provides the following function, which is called when we want to start a new private session:

```
int chat_offer_start(ChatContextPtr ctx, ChatMessage **msgToSend);
```

Listing 6.3: Offer specific interface

The code of the offer component is listed in appendix B′.

DSKE

The states of the DSKE sub-protocol are defined as following:

```
typedef enum {
    CHAT_DSKESTATE_NONE,
    CHAT_DSKESTATE_AWAITING_KEYS,
    CHAT_DSKESTATE_FINISHED
} ChatDSKEState;
```

Listing 6.4: DSKE states

The DSKE sub-protocol component uses a component called *chat_dake* which implements the denAKE protocol.

The code of the DSKE component is listed in appendix Γ′.

GKA

The states of the GKA sub-protocol are defined as following:

```
typedef enum {
    CHAT_GKASTATE_NONE,
    CHAT_GKASTATE_AWAITING_UPFLOW,
    CHAT_GKASTATE_AWAITING_DOWNFLOW,
    CHAT_GKASTATE_FINISHED
} ChatGKAState;
```

Listing 6.5: GKA states

The code of the GKA component is listed in appendix Δ′.

Attest

The states of the Attest sub-protocol are defined as following:

```
typedef enum {
    CHAT_ATTESTSTATE_NONE,
    CHAT_ATTESTSTATE_AWAITING,
    CHAT_ATTESTSTATE_FINISHED
} ChatAttestState;
```

Listing 6.6: Attest states

The code of the Attest component is listed in appendix E′.

Communication

Communication has no internal information, so its interface contains only the last two functions of listing 6.1.

In addition, it provides the following function, used to broadcast an encrypted and signed message in an already set up session:

```
int chat_communication_broadcast(ChatContextPtr ctx, const char
    *message, ChatMessage **msgToSend);
```

Listing 6.7: Communication specific interface

The code of the Communication component is listed in appendix ΣT′.

Shutdown

The states of the Shutdown sub-protocol are defined as following:

```
typedef enum {
    CHAT_SHUTDOWNSTATE_NONE,
    CHAT_SHUTDOWNSTATE_AWAITING_SHUTDOWNS,
    CHAT_SHUTDOWNSTATE_AWAITING_DIGESTS,
    CHAT_SHUTDOWNSTATE_AWAITING_ENDS,
```

```
    CHAT_SHUTDOWNSTATE_FINISHED
} ChatShutdownState;
```

Listing 6.8: Shutdown states

In addition to the general sub-protocol interface shown in listing 6.1, Shutdown provides the following functions, used to send specific shutdown messages:

```
int chat_shutdown_send_shutdown(ChatContextPtr ctx, ChatMessage
    **msgToSend);
int chat_shutdown_send_digest(ChatContextPtr ctx, ChatMessage
    **msgToSend);
int chat_shutdown_send_end(ChatContextPtr ctx, ChatMessage
    **msgToSend);
int chat_shutdown_release_secrets(ChatContextPtr ctx, ChatMessage
    **msgToSend);
```

Listing 6.9: Shutdown specific interface

The code of the Shutdown component is listed in appendix Z′.

6.6.3 Infrastructure components

Each of the following infrastructure components incorporates a First-Class ADT implementation accompanied by relevant functions:

chat_context: It incorporates a First-Class ADT with handle type *ChatContextPtr* that models the context of a chat room. Each instance contains details of the chat room (user's account, protocol in use, list of participants, etc.), handles for each sub-protocol info and details of the mpOTR session (session ID, ehpemeral signing key, shared secret, long-term identity key, etc.). It also provides functions to add or find a context in a list of contexts.

chat_message: It provides functions to create each different type of mpOTR message. Each mpOTR message is modeled as a structure of type *ChatMessage*. It also provides functions to serialize and parse a serialized mpOTR message.

chat_participant: It incorporates a First-Class ADT with handle type *ChatParticipantPtr* that models a participant of the chat room. It also provides functions to add of find a participant in a list of participants.

chat_id_key: It incorporates a First-Class ADT with handle type *ChatIdKeyPtr* that models a longterm identity key of the user. The actual type of the identity key may be any type that implements the interface shown in listing 6.10. It also provides functions to generate a new identity key for an account, and to find an identity key in a list of identity keys. Finally, it provides functions to read an identity key list from or write to a file.

```
struct ChatInternalKeyOps{
    ChatInternalKeyPtr (*generate)(void);
    int (*serialize)(ChatInternalKeyPtr, gcry_sexp_t *);
    ChatInternalKeyPtr (*parse)(gcry_sexp_t);
    unsigned char *
    (*fingerprint_create)(ChatInternalKeyPtr);
    void (*free)(ChatInternalKeyPtr);
};
```

Listing 6.10: Internal key interface

chat_dh_key: A First-Class ADT with handle type *ChatDHKeyPtr* that models a Diffie–Hellman key pair and implements the internal key interface listed in listing 6.10.

chat_fingerprint: It incorporates a First-Class ADT with handle type *OtrlChatFingerprintPtr* that models a fingerprint of a participant's public identity key. It also provides functions to find, add or remove a fingerprint in a list of fingerprints. Finally, it provides functions to read a fingerprint list from or write to a file.

chat_pending: It incorporates a First-Class ADT with handle type *ChatPendingPtr* that models a fingerprint of a participant's public identity key. Each instance contains the sender username and the message string.

chat_event: It incorporates a First-Class ADT with handle type *OtrlChatEventPtr* that models a fingerprint of a participant's public identity key. Each instance contains the event type and the type specific data. It provides an event creating function for each type of event.

chat_info: It incorporates a First-Class ADT with handle type *OtrlChatInfoPtr* that models a chat room descriptor that contains chat room specific information to be sent to the client application.

list: It incorporates a doubly linked list implementation and a list iterator implementation. The list is implemented as a First-Class ADT with handle type *OtrlListPtr*. The iterator is also implemented as a First-Class ADT with handle type *OtrlListIteratorPtr*.

6.6.4 Functional components

These are the components that provide low level functions. There are three of them:

chat_sign: It provides functions for signature generation and verification.

chat_enc: It provides functions for encryption and decryption.

chat_serial: It provides functions for serialization and parsing of several variable types used in mpOTR messages.

6.7 Application Programming Interface

6.7.1 The *OtrlUserState* instance handle

The OTR library uses a handle to associate internal protocol data with the client. This is a variable of type *OtrlUserState*. We used this handle to associate mpOTR Protocol internal data, too. Regarding the mpOTR implementation, it encapsulates the list of private identity keys, the list of known fingerprints and a list of mpOTR session descriptors. The *OtrlUserState* instance handle will be passed as an argument to various mpOTR library API endpoints.

Most clients will only need one instance handle, but in a case of a client implementing a multi-user context there may be the need for more instance handles.

The following functions are provided to create a new handle and to free an already created *OtrlUserState*:

```
OtrlUserState otrl_userstate_create(void);
void otrl_userstate_free(OtrlUserState us);
```

Listing 6.11: OtrlUserState interface

The client should free every created *OtrlUserState* handle when he is finished with them.

6.7.2 The *otrl_chat_token_t* chat room identifier

When the client wants to call chat room specific library functions, it should provide a value that identifies the specific chat room. The client should assure that only one chat room corresponds to a specific value and vice versa. We call this type of value *otrl_chat_token_t* which is defined:

```
typedef int otrl_chat_token_t;
```

LISTING 6.12: otrl_chat_token_t definition

6.7.3 The *OtrlChatInfoPtr* chat room descriptor

OtrlChatInfoPtr is a First-Class ADT used to encapsulate information about a specific chat room. The following interface is provided for the client:

```
char * otrl_chat_info_get_accountname(OtrlChatInfoPtr info);
char * otrl_chat_info_get_protocol(OtrlChatInfoPtr info);
otrl_chat_token_t otrl_chat_info_get_chat_token(OtrlChatInfoPtr
    info);
OtrlChatPrivacyLevel
    otrl_chat_info_get_privacy_level(OtrlChatInfoPtr info);
```

LISTING 6.13: OtrlChatInfoPtr First-Class ADT interface

The *OtrlChatPrivacyLevel* is defined as follows:

```
typedef enum {
    OTRL_CHAT_PRIVACY_LEVEL_NONE,
    OTRL_CHAT_PRIVACY_LEVEL_UNVERIFIED,
    OTRL_CHAT_PRIVACY_LEVEL_PRIVATE,
    OTRL_CHAT_PRIVACY_LEVEL_FINISHED,
    OTRL_CHAT_PRIVACY_LEVEL_UNKNOWN
} OtrlChatPrivacyLevel;
```

LISTING 6.14: OtrlChatPrivacyLevel definition

6.7.4 The *OtrlMessageAppOps* callbacks structure

In order for a client to use OTR Library, it must provide a structure containing pointers to functions that must be defined by the client but called by the library. The type of structure is called *OtrlMessageAppOps*. We added callbacks needed for mpOTR in this structure, too. Regarding the mpOTR implementation the following callbacks were added:

```
int (*chat_inject_message)(void *opdata, const OtrlChatInfoPtr
    info, const char *message);
void (*chat_handle_event)(void *odata, const OtrlChatInfoPtr
    info, const OtrlChatEventPtr event);
void (*chat_display_notification_cb)(void *opdata, const
    OtrlChatInfoPtr info, const char *notification);
char **(*chat_get_participants)(void *opdata, const
    OtrlChatInfoPtr info, unsigned int *size);
void (*chat_privkey_create)(void *opdata, const char
    *accountname, const char *protocol);
```

```
void (*chat_fingerprints_write)(void *opdata);
void (*chat_privkeys_write)(void *opdata);
void (*chat_info_refresh)(void *opdata, const OtrlChatInfoPtr
    info);
```

Listing 6.15: mpOTR callbacks in OtrlMessageAppOps

chat_inject_message: Broadcasts *message* to the chatroom defined by *info*. Returns *0* on success, non-zero on error.

chat_handle_event: Reacts to *event* that happened in the chatroom defined by *info*. The events are described in section 6.7.11.

chat_display_notification_cb: Displays *notification* regarding the chatroom defined by *info*.

chat_get_participants: Provides the list of the participants' usernames that are currently in the chatroom defined by *info*. Returns a pointer to an array of the usernames along with the number of participants in the output parameter **size* on success, *NULL* on error.

chat_privkey_create: Invokes the library's function that creates a new private key for this *accountname* and *protocol*, providing it with the proper file descriptor. The invoked function is described in 6.7.8.

chat_fingerprints_write: Invokes the library's function that writes the known fingerprints to a file, providing it with the proper file descriptor. The invoked function is described in section 6.7.10.

chat_privkeys_write: Invokes the library's function that writes the private identity keys to a file, providing it with the proper file descriptor. The invoked function is described in 6.7.8.

chat_info_refresh: Reacts to a change regarding the status of the chatroom defined by *info*.

6.7.5 Starting private session

To start a private session the client should use:

```
int otrl_chat_protocol_send_query(OtrlUserState us, const
    OtrlMessageAppOps *ops, const char *accountname, const char
    *protocol, otrl_chat_token_t chat_token);
```

Listing 6.16: The private session initiation function

The following parameters should be provided:

us: The instance handle.

ops: The callbacks structure.

accountname: The identifying name for the account in use.

protocol: The identifying name for the protocol in use.

chat_token: The chat room identifier.

6.7.6 Ending a private session

To end a private session the client should use:

```
int otrl_chat_protocol_shutdown(OtrlUserState us, const
    OtrlMessageAppOps *ops, const char *accountname, const char
    *protocol, otrl_chat_token_t chat_token);
```

LISTING 6.17: The private session ending function

The following parameters should be provided:

us: The instance handle.

ops: The callbacks structure.

accountname: The identifying name for the account in use.

protocol: The identifying name for the protocol in use.

chat_token: The chat room identifier.

6.7.7 Handling messages

Since the client knows nothing about active mpOTR sessions and the structure of mpOTR messages, it should pass every message to mpOTR functions before sending or after receiving them. The mpOTR will decide if they should be handled by the library or not. The following two functions are provided.

Upon receiving a message in a chatroom and before displaying it to the user the client application should call:

```
int otrl_chat_protocol_receiving(OtrlUserState us, const
    OtrlMessageAppOps *ops, void *opdata, const char *accountname,
    const char *protocol, const char *sender, otrl_chat_token_t
    chat_token, const char *message, char **newmessagep);
```

LISTING 6.18: The received messages handling function

The following parameters should be provided:

us: the instance handle

ops: the callbacks structure

opdata:

accountname: the identifying name for the account in use

protocol: the identifying name for the protocol in use

sender: the username of the message sender

chat_token: the chat room identifier

message: the received message

**newmessagep*: output parameter that contains contains a newly allocated string that should be displayed to the user instead of message, or *NULL* if no modification is needed.

If *0* is returned, *message* was an ordinary, non-OTR message, which should be delivered to the user without modification. If *1* is returned, the message was handled by mpOTR library. In this case, if **newmessagep* is *NULL* then nothing should be displayed to the user. Else **newmessagep* should be displayed to the user intead of the received one and be freed afterwards.

Before sending a user's message the client application should call:

```
int otrl_chat_protocol_sending(OtrlUserState us, const
    OtrlMessageAppOps *ops, void *opdata, const char *accountname,
    const char *protocol, const char *message, otrl_chat_token_t
    chat_token, char **messagep);
```

Listing 6.19: The sending messages handling function

The following parameters should be provided:

us: the instance handle

ops: the callbacks structure

opdata:

accountname: the identifying name for the account in use

protocol: the identifying name for the protocol in use

sender: the username of the message sender

message: the message to be sent

chat_token: the chat room identifier

**messagep*: output parameter that contains contains a newly allocated string that should be sent instead of *message*, or *NULL* if no modification is needed.

Returns a non-zero in case of error. In this case, nothing is safe to be sent. If *0* is returned then the client should check **messagep*. If **messagep* is *NULL* then *message* should be sent unmodified. Else **messagep* should be sent to the chat room intead of *message* and be freed afterwards.

6.7.8 Private identity key management

The following functions are provided for the management of the private identity keys:

```
int otrl_chat_protocol_id_key_read_file(OtrlUserState us, FILE
    *privf);
int otrl_chat_protocol_id_keys_write_file(OtrlUserState us, FILE
    *privf);
int otrl_chat_protocol_id_key_generate_new(OtrlUserState us,
    const OtrlMessageAppOps *ops, const char *accountname, const
    char *protocol);
OtrlListPtr otrl_chat_protocol_id_key_list_create(OtrlUserState
    us);
**messagep);
```

Listing 6.20: Private identity key management functions

otrl_chat_protocol_id_key_read_file: Loads the private identity key list from the specified file to the memory. It should be called once before using the mpOTR Library for instance handle *us*.

otrl_chat_protocol_id_keys_write_file: Writes the private identity key list loaded in the memory, to the specified file. It is meant to be called only inside the proper callback function, see section 6.7.4.

otrl_chat_protocol_id_key_generate_new: Creates a new private identity key for the specified *accountname* and *protocol*.

otrl_chat_protocol_id_key_list_create: Creates and returns a list containing elements of type *OtrlChatIdKeyInfoPtr*. Each element contains information about each private identity key.

The structure pointer *OtrlChatIdKeyInfoPtr* is defined as follows:

```
typedef struct OtrlChatIdKeyInfo * OtrlChatIdKeyInfoPtr;

struct OtrlChatIdKeyInfo {
    char *accountname;
    char *protocol;
    char *fingerprint_hex;
};
```

Listing 6.21: OtrlChatIdKeyInfoPtr definition

6.7.9 The *OtrlChatFingerprintPtr* Frist-Class ADT

The fingerprints are implemented as First-Class ADT. The following interface is provided:

```
char *otrl_chat_fingerprint_bytes_to_hex(const unsigned char
    *fingerprint);
size_t chat_fingerprint_size();
char *
    otrl_chat_fingerprint_get_accountname(OtrlChatFingerprintPtr
    fnprnt);
char * otrl_chat_fingerprint_get_protocol(OtrlChatFingerprintPtr
    fnprnt);
char * otrl_chat_fingerprint_get_username(OtrlChatFingerprintPtr
    fnprnt);
unsigned char *
    otrl_chat_fingerprint_get_bytes(OtrlChatFingerprintPtr fnprnt);
```

Listing 6.22: OtrlChatFingerprintPtr Frist-Class ADT interface

otrl_chat_fingerprint_bytes_to_hex: Converts fingerprint bytes to human-readable form. The result should be freed by the caller.

otrl_chat_fingerprint_get_accountname: Returns the username of the user that knows this fingerprint.

otrl_chat_fingerprint_get_protocol: Returns the protocol used when this fingerprint was met.

otrl_chat_fingerprint_get_username: Returns the username of the user who has an identity key with that fingerprint.

otrl_chat_fingerprint_get_bytes: Returns the actual bytes of the fingerprint.

6.7.10 Known fingerprints management

The following functions are provided regarding the management of the known fingerprints:

```
int otrl_chat_protocol_fingerprints_read_file(OtrlUserState us,
    FILE *fingerfile);
int otrl_chat_protocol_fingerprints_write_file(OtrlUserState us,
    FILE *fingerfile);
void otrl_chat_protocol_fingerprint_verify(OtrlUserState us,
    const OtrlMessageAppOps *ops, OtrlChatFingerprintPtr fnprnt);
void otrl_chat_protocol_fingerprint_forget(OtrlUserState us,
    const OtrlMessageAppOps *ops, OtrlChatFingerprintPtr fnprnt);
```

Listing 6.23: Known fingerprints management functions

otrl_chat_protocol_fingerprints_read_file: Loads the known fingerprints list from the specified file to the memory. It should be called once before using the mpOTR Library for instance handle *us*.

otrl_chat_protocol_fingerprints_write_file: Writes the known fingerprints list loaded in the memory, to the specified file. It is meant to be called only inside the proper callback function, see section 6.7.4.

otrl_chat_protocol_fingerprint_verify: Marks the specified fingerprint as verified.

otrl_chat_protocol_fingerprint_forget: Removes the specified fingerprint from the known fingerprints list.

6.7.11 Events

Events are implemented as a First-Class ADT called *OtrlChatEventDataPtr*. The following interface is provided:

```
OtrlChatEventType otrl_chat_event_get_type(OtrlChatEventPtr
    event);
OtrlChatEventDataPtr otrl_chat_event_get_data(OtrlChatEventPtr
    event);
```

Listing 6.24: OtrlChatEventDataPtr First-Class ADT interface

otrl_chat_event_get_type: Returns an *OtrlChatEventType* indicating the type of the event.

otrl_chat_event_get_data: Returns the type specific data of the event if any, else returns NULL.

Event types

The following event types are defined:

```
typedef enum {
    OTRL_CHAT_EVENT_OFFER_RECEIVED,
    OTRL_CHAT_EVENT_STARTING,
    OTRL_CHAT_EVENT_STARTED,
    OTRL_CHAT_EVENT_UNVERIFIED_PARTICIPANT,
    OTRL_CHAT_EVENT_PLAINTEXT_RECEIVED,
    OTRL_CHAT_EVENT_PRIVATE_RECEIVED,
    OTRL_CHAT_EVENT_CONSENSUS_BROKEN,
```

```
    OTRL_CHAT_EVENT_FINISHED
} OtrlChatEventType;
```

Listing 6.25: OtrlChatEventType definition

OTRL_CHAT_EVENT_OFFER_RECEIVED: Emitted when we received an offer. Contains intenal data of type *OtrlChatEventParticipantDataPtr*.

OTRL_CHAT_EVENT_STARTING: Emitted when the protocol attempts to start a private session. Contains no internal data.

OTRL_CHAT_EVENT_STARTED: Emitted when the private session has started. Contains no internal data.

OTRL_CHAT_EVENT_UNVERIFIED_PARTICIPANT: Emitted when the private session has started with an unverified participant in it. Contains intenal data of type *OtrlChatEventParticipantDataPtr*.

OTRL_CHAT_EVENT_PLAINTEXT_RECEIVED: Emitted when we receive a plaintext message while in a private session. Contains intenal data of type *OtrlChatEventMessageDataPtr*.

OTRL_CHAT_EVENT_PRIVATE_RECEIVED: Emitted when we receive a private message while NOT in a private session. Contains intenal data of type *OtrlChatEventParticipantDataPtr*.

OTRL_CHAT_EVENT_CONSENSUS_BROKEN: Emitted when there was no consensus with a participant. Contains intenal data of type *OtrlChatEventParticipantDataPtr*.

OTRL_CHAT_EVENT_FINISHED: Emitted when a private session was finished. Contains no internal data.

Internal event data

There are two different types of internal event data implemeneted as First-Class ADTs.

The first one is the *OtrlChatEventParticipantDataPtr* with the following interface:

```
char * otrl_chat_event_participant_data_get_username(
    OtrlChatEventParticipantDataPtr data);
```

Listing 6.26: OtrlChatEventDataPtr First-Class ADT interface

The second one is the *OtrlChatEventMessageDataPtr* with the following interface:

```
char * otrl_chat_event_message_data_get_username(
    OtrlChatEventMessageDataPtr data);
char * otrl_chat_event_message_data_get_message(
    OtrlChatEventMessageDataPtr data);
```

Listing 6.27: OtrlChatEventDataPtr First-Class ADT interface

Chapter 7

The mpOTR Plugin

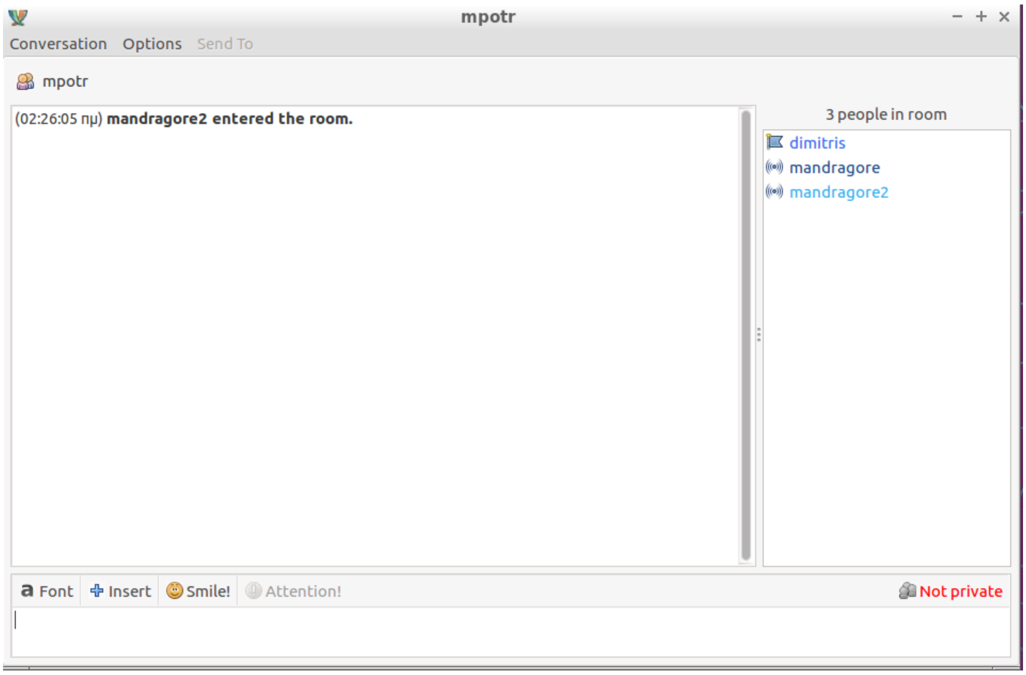
In addition to the library we also extended the otr pidgin plugin's functionality. It now uses the extended capabilities of libotr in order to provide private multi-party chatrooms.

Pidgin is an Instant Messaging (IM) client that is compatible with a wide range of IM protocols. Since our protocol is protocol agnostic[1] pidgin users can readily chat securely with their existing contacts.

7.1 The plugin workflow

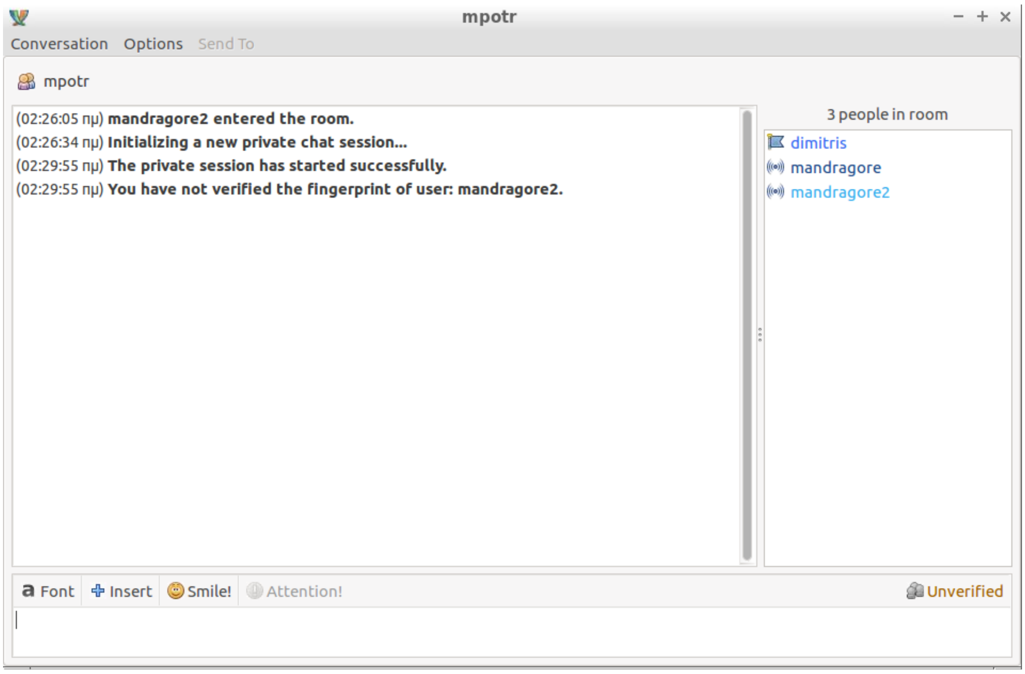
Allow us now to present in summary the workflow of the plugin.

This is what a pidgin chat conversation looks like when no mpOTR session is taking place. Notice the mpOTR button on the lower right corner, similar to the OTR button.

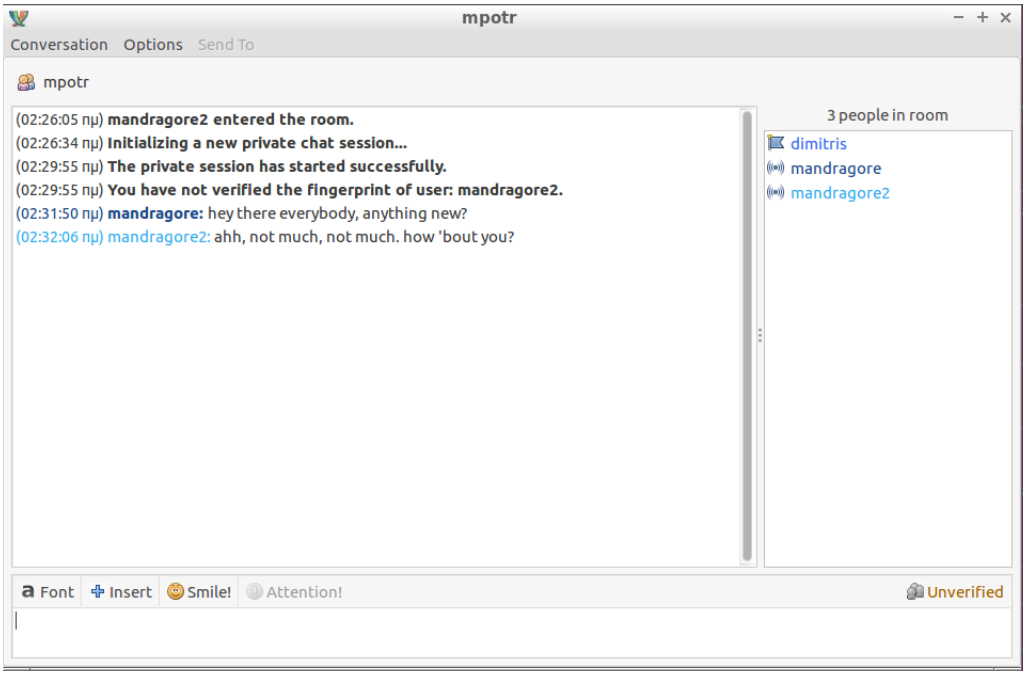


By clicking on the mpOTR button a user has the option to start a private conversation. If he chooses to do so this is what he sees.

[1]This is not wholly true. Our implementation of mpOTR assumes that it can send messages of arbitrary length. This is not true in all IM protocols, IRC being one example.

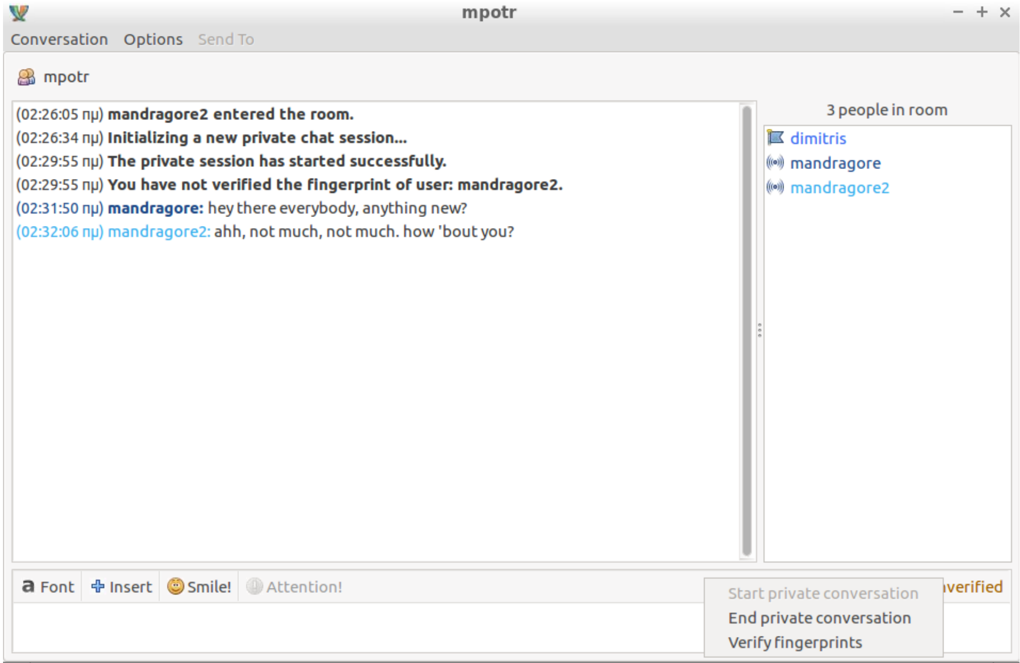


And when some texts are exchanged.

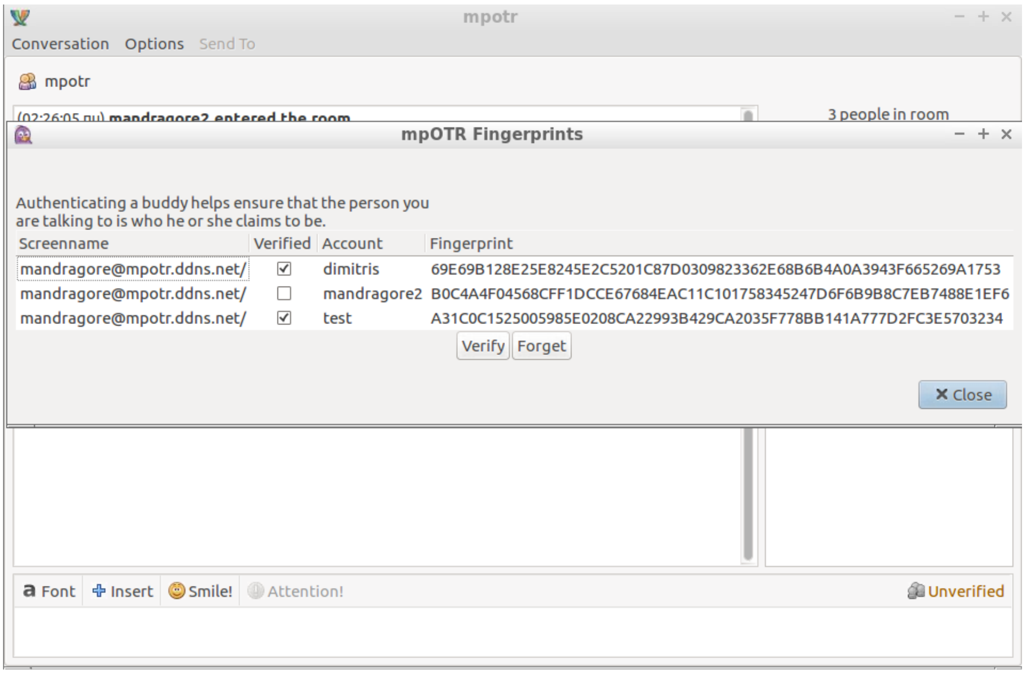


However, our user (mandragore) hasn't verified another user (mandragore2). This means that the conversation is unverified. This is presented to the user in two ways. First the mpOTR button has a yellow colour and states that the conversation is "Unverified". And then, the message "You have not verified user: mandragore2". This message will be displayed for every unverified user.

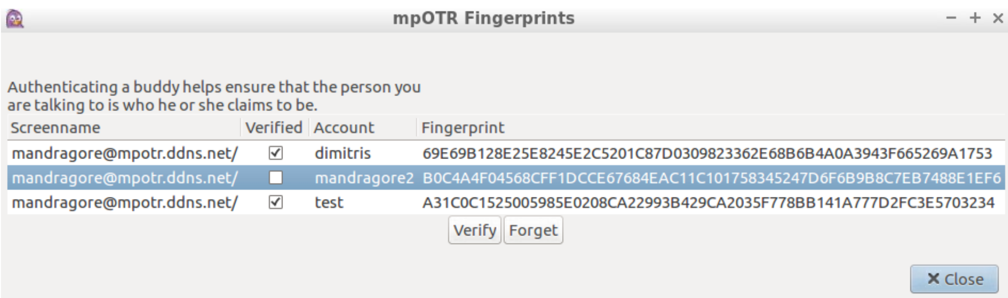
In order to verify the user mandragore2, our user clicks on the mpOTR button. Notice how the "Start private conversation" option is now disabled.

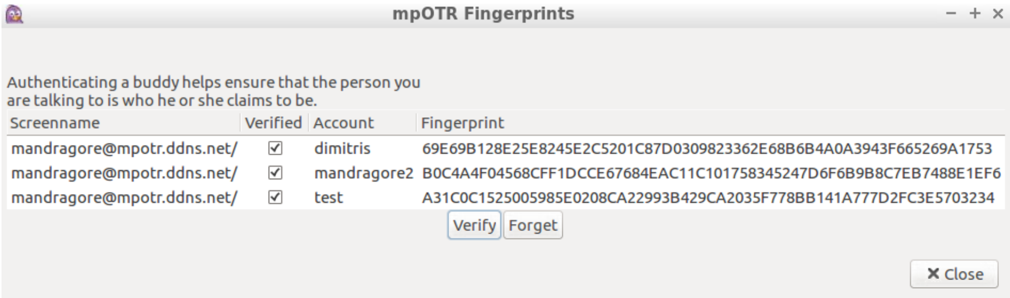


If he clicks the "Verify fingerprints" option this window opens.

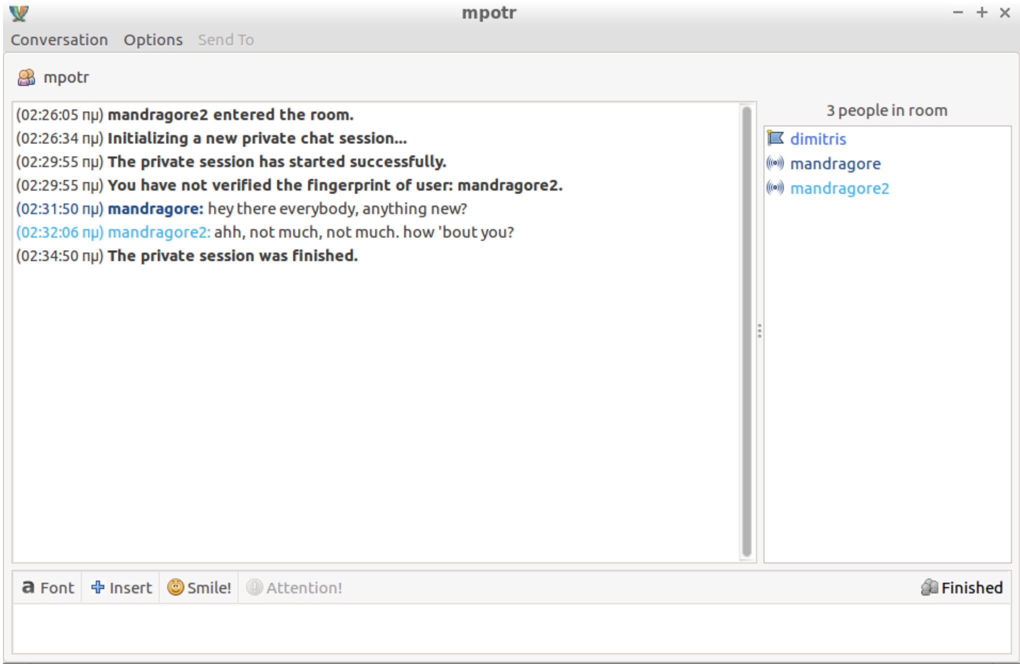


In this window the user can click on the user he wants to verify and (after he checks the fingerprint) click on the "Verify" button. The selected user will now be verified.

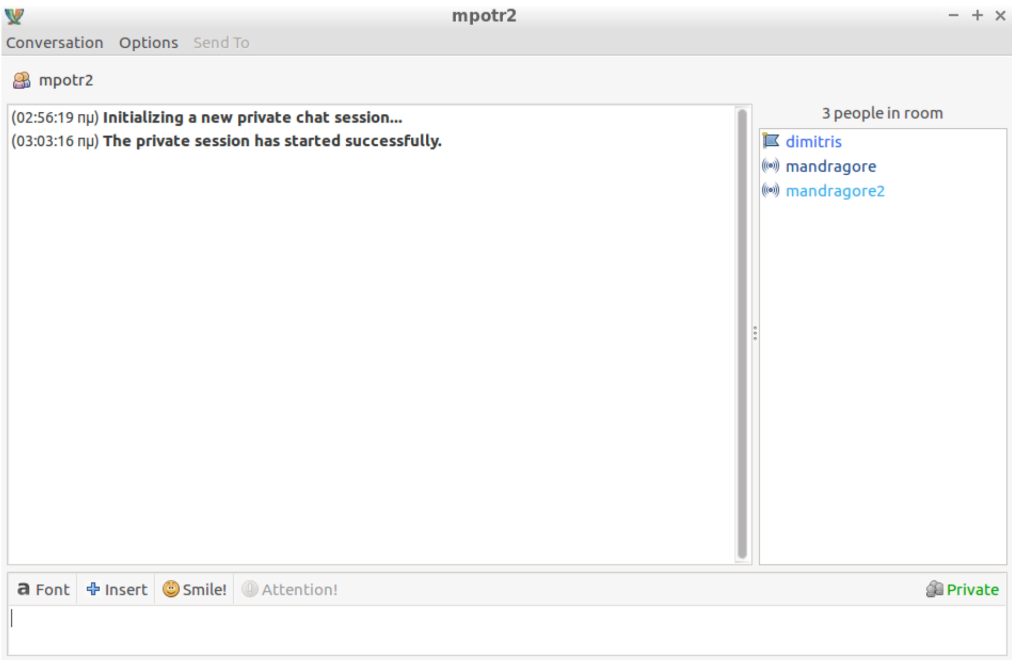




To end the conversation the user clicks on the mpOTR button again and selects the ”End private conversation” option.



Now if the user starts another private conversation the new session will be characterised as ”Private”.



Chapter 8

Related Work

As we have already stated, end-to-end encrypted instant messaging is a very trending topic in the crypto community. Consequently, a plethora of protocols and implementations achieving the above goal exist, the majority of them handling the two-party case.

Here we will present a brief collection of the aforementioned protocols. Our goal is not to be exhaustive, but rather to give credit to the authors of those protocols. Either because their ideas gave us inspiration, or because we experienced first hand the difficulties of implementing and designing secure communication applications, and would like to acknowledge their contributions.

8.1 Two-party Protocols

First we will take a look at protocols providing conversations between two parties.

8.1.1 Off-the-Record Messaging

This protocol could not be missing from this section. It is designed by the same team which authored the original Multi-part OTR paper [7].

As far as we know it is the first complete protocol to provide end-to-end encrypted Instant Messaging with strong cryptography. It set the ground for many other protocols and algorithms, like the axolotl key ratchet, which later evolved into the Signal protocol.

8.1.2 Vuvuzela

"*We kill people based on metadata*"

Michael Hayden, former NSA director

While end-to-end encryption is an essential component for any privacy protecting protocol, it is not perfect on its own. The metadata can reveal a bunch of crucial information that can be used to even determine the contents of encrypted data.

Vuvuzela [6] is a protocol that provides strong metadata privacy that scales. It utilizes onion encryption (like TOR) to hide metadata. Messages are sent in rounds, and noise packets are injected in the network in order to defend against traffic analysis attacks.

8.2 Multi-party Protocols

And now let's see some multi-party protocols.

8.2.1 Flute

Flute is a secure multiparty messaging protocol, currently available as a weechat plugin for irc chatrooms. It provides end-to-end encryption, but does not offer other desirable properties like deniability or chatroom message consistency.

Flute does not provide everything that is needed for a multi-party protocol to be completely secure. It's simplicity however, allowed it to have a quick implementation, and also provides the challenge to figure out what protocol are crucial in multi-party messaging, and must be added, or do not really enhance the security and should better be left out.

8.2.2 Signal

Signal, developed by Open Whisper Systems, is currently the most complete solution to the multi-party messaging problem. It is a robust protocol providing all the necessary properties like *confidentiality, authentication, deniability* and *forward secrecy*. It does not provide *transcript consistency* but the double-ratchet it uses provides some resistance against reordering attacks.

It is available as an Android and iOS application, and for two party conversation can be used through the chrome browser. Independently from its authors various applications also use it. What'sApp, Viber, and Facebook Messenger to name a few, are some of the applications that use the Signal protocol to provide secure chats. Cryptocat is a firefox plugin that closely follows the Signal approach and is completely open source.

Chapter 9

Future Work

In previous chapters we addressed several problems regarding specific parts of our mpOTR construction. While our construction is fully functional and secure, solving these problems would enhance the protocol's privacy and/or usabilty. In this chapter we fully describe each of them and suggest possible solutions.

9.1 Message Fragmentation

Some networks may have a message length limitation that is too small to contain an mpOTR message. To solve this problem, OTR has utilized a fragmentation mechanism since version 2.

The OTR fragmentation mechanism is intuitive. On the sender's side, messages are splitted into a sufficient number of pieces N, so that every fragment does not exceed the specified length limit. Each fragment contains a sequence number k, the value of N and the actual piece. The number k indicates the position of the piece in the whole message, starting from 1 and ending to N. On the recipient's side, the fragments are accumulated so that after receiving the piece with its sequence number k equal to N the whole message can be reconstructed.

This fragmentation mechanism works properly as long as the fragments are delivered in-order. In case of out-of-order delivery the algorithm implements a fragment forgetting strategy that finally rejects the message. A more severe problem arises when fragments from different messages are delivered intermixed. Since each fragment contains no information identifying the message it belongs to, the only distinctive information is the number N. In case N contained in a newly received fragment is different than the one contained in the so far accumulated fragments, the so far accumulated fragments are forgotten. But what happens if fragments from different messages splitted into the same number of pieces N are delivered in an intermixed order? Hopefully, the sequence numbers k won't form a valid sequence. In the extreme scenario the latter happens a non-valid message will be reconstructed.

The problems described above are actually unlikely to happen in a two-party communication context. But such a mechanism in a multi-party context would be a bad choice. Even if fragments are accumulated separately for each sender, there is a sporting chance that fragments from different messages sent during the setup procedure will be delivered intermixed. Rejecting such a message would require a restart of the whole setup procedure, and that could possibly occur indefinitely.

9.2 Message consistency in constant space

In [7] a straightforward approach is followed. In order to check if each participant has received the same set of messages all the messages from each user must be stored, and during the shutdown phase be lexicographically ordered and hashed. While this achieves our purpose it requires $O(M)$ space where M is the number of messages.

We believe that the same effect can be achieved in constant space by using cryptographic accumulators. One can find more about this primitive in [3].

Since such accumulators are collision free but at the same time quasi-commutative, they are ideal for our purposes. We can feed the accumulator with the incoming messages in whichever order they arrive at each participant. The quasi-commutative property guarantees that if two participants have received the same set of messages then their accumulators will arrive at the same value in the end.

Thus, we have removed the need to store the messages in order to sort them during the shutdown phase. We only need to store the value of the accumulator which, of course, is constant.

9.3 Longterm public identity key verification via OTR

The authentication of participants' identities premises that each participant has verified the others' longterm public identity keys. This means that each pair of participants should exchange their longterm public identity keys using some already authenticated communication channel. For now, we assume that this exchange is done manually, either in person or using other authenticated channels like simple OTR, GPG-signed emails, etc.

Although our implementation of the mpOTR Protocol is integrated with the OTR library, the longterm identity keys used in mpOTR are completely different than the ones used in OTR. That's because in mpOTR the longterm identity keys are Diffie-Hellman keys while in OTR they are DSA keys. Having to verify two different keys for the same person could prove confusing for casual users.

One possible solution for this problem is to use an already verified OTR IM channel in order to exchange the mpOTR longterm public identity keys behind the scenes.

9.4 Message Ordering with OldBlue Protocol

In chapter 3 we discussed the problem of message ordering. We also promised that we would investigate a possible solution to that problem. In particular we will talk about the OldBlue protocol. As we shall see this protocol guarantees that the received messages are causally ordered.

9.4.1 Why causal ordering

To understand why the causal ordering is the best we can get let's see what we can and cannot do: First note that a multi-party chat room is a distributed environment. This means that there is no central "authority" which can decide if a message came before another.

Also, since we are in a zero trust setting we cannot utilise any values over which the participants have full control, like their local clocks for example.

The only information for which we must trust the other participants is what messages they have seen. We have no other option on this one as we cannot know if an attacker has actually stopped messages from getting to them or if they are dishonest and lie to us.

As a result the only information about the history of a received message that we can trust (*must* trust actually) is this: We can only know what message a user admits she has seen, before authoring the received message. In other words the only thing we can know is which messages might have *caused* the received message.

9.4.2 The parent graph

Using this information on the causal ordering of messages we can construct the parent graph. This graph will contain all the information about which messages came after some others. Now lets take a look on the structure of this graph.

Firs of all it is directed. It retains information on which messages came after other messages. Since this property is not symmetric our parent graph has directions on its edges.

It is also obvious that this graph is acyclic. No cycles can exist in this graph since a message cannot possibly cause any of its ancestors. In other words the edges will always point from the beginning of the conversation towards its end.

In an ideal world, this graph would be a line. Every user would see all the previous messages before sending a message, and no two messages would be sent simultaneously. However this is not the case generally. Two users may have seen the same set of parent messages and choose to send a message at the same time. As a result the parent graph will fork, as the same message now has two children.

9.4.3 Distributed Parent Graph

As we said a multi-party chatroom is a distributed environment. This means that each participant stores and builds a local copy of the parent graph. To do that each participant attach to each message he sends some information of his local copy so that other participants can find out where they should place his message.

The naive solution would be to include the whole current parent graph in his message but that is obviously infeasible. Instead much less information is needed. The actual information that he must send is all the messages in his graph that do not have any children. This is called the "front" of the graph.

This information is transmitted by appending a list of the hashes of all the "front" messages, to the message to be sent.

9.4.4 Dangling messages

If no re-ordering of the messages occurs then everything works fine. How do we handle reordered messages however?

The protocol handles two sets of messages. One is called the delivered set, the other is called undelivered. When a message is received we check if all of its parents are in the delivered set. If not then the message is added in the undelivered set and waits there.

If the parents are delivered then the message itself is inserted in the delivered set and displayed to the user. After that happens the protocol iterates over the undelivered set and checks if any message is now deliverable (meaning that all of its parents are now in the delivered set). If it is then, the message is added in the delivered set and displayed to the user. After that it iterates again over the undelivered set and repeats until there are no deliverable messages.

9.5 Group Encryption Key Ratcheting

Plain OTR has a property called *future secrecy*. This means that if for some reason the shared secret is compromised, only a few messages in the chat will be revealed to the attacker. In fact if both of the two participants send a messages after the compromise, then a new key will be generated and the old compromised secret will be useless.

This does not happen in our protocol. After the GKA is finished the shared secret remains the same until the shutdown phase.

A similar result can be achieved if we run various GKAs one after the other in the background. We could attach the required upflow and downflow messages in chat messages sent by users during the communications protocol. If a particular user is away or doesn't participate actively in the chat by authoring messages an chat message can be sent, using a timer interrupt, which will contain the data required for the GKA to continue. This way the group secret could be ratcheted.

A side-effect of this approach is that the ratcheting of the key will work as a central "clock" of the chatroom. Messages sent before the change of the secret will no longer be readable by the participants. This way an attacker would not be able to reorder messages encrypted using two different secrets.

Appendix A′

Protocol source code

```
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros
    <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos
    <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
    02110-1301 USA
 */

#ifndef CHAT_PROTOCOL_H_
#define CHAT_PROTOCOL_H_

int otrl_chat_protocol_id_key_read_file(OtrlUserState us, FILE *privf);
int otrl_chat_protocol_id_keys_write_file(OtrlUserState us, FILE *privf);
int otrl_chat_protocol_id_key_generate_new(OtrlUserState us, const
    OtrlMessageAppOps *ops, const char *accountname, const char *protocol);
OtrlListPtr otrl_chat_protocol_id_key_list_create(OtrlUserState us);

int otrl_chat_protocol_fingerprints_read_file(OtrlUserState us, FILE
    *fingerfile);
int otrl_chat_protocol_fingerprints_write_file(OtrlUserState us, FILE
    *fingerfile);
void otrl_chat_protocol_fingerprint_verify(OtrlUserState us, const
    OtrlMessageAppOps *ops, OtrlChatFingerprintPtr fnprnt);
void otrl_chat_protocol_fingerprint_forget(OtrlUserState us, const
    OtrlMessageAppOps *ops, OtrlChatFingerprintPtr fnprnt);

int chat_protocol_reset(ChatContextPtr ctx);

 int otrl_chat_protocol_receiving(OtrlUserState us, const
    OtrlMessageAppOps *ops,
    void *opdata, const char *accountname, const char *protocol,
    const char *sender, otrl_chat_token_t chat_token, const char *message,
    char **newmessagep, OtrlTLV **tlvsp);

 int otrl_chat_protocol_sending(OtrlUserState us,
    const OtrlMessageAppOps *ops,
    void *opdata, const char *accountname, const char *protocol,
    const char *message, otrl_chat_token_t chat_token, OtrlTLV *tlvs,
    char **messagep, OtrlFragmentPolicy fragPolicy);

 int otrl_chat_protocol_send_query(OtrlUserState us,
        const OtrlMessageAppOps *ops,
```

```
        const char *accountname, const char *protocol,
        otrl_chat_token_t chat_token, OtrlFragmentPolicy fragPolicy);

 int otrl_chat_protocol_shutdown(OtrlUserState us, const OtrlMessageAppOps
    *ops,
        const char *accountname, const char *protocol, otrl_chat_token_t
    chat_token);

#endif /* CHAT_PROTOCOL_H_ */
```

Listing A′.1: chat_protocol.h

```
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros
    <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos
    <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
    02110-1301 USA
 */

#include <gcrypt.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "b64.h"
#include "chat_attest.h"
#include "chat_communication.h"
#include "chat_context.h"
#include "chat_dh_key.h"
#include "chat_dske.h"
#include "chat_event.h"
#include "chat_fingerprint.h"
#include "chat_gka.h"
#include "chat_id_key.h"
#include "chat_info.h"
#include "chat_message.h"
#include "chat_offer.h"
#include "chat_participant.h"
#include "chat_pending.h"
#include "chat_shutdown.h"
#include "chat_sign.h"
#include "chat_types.h"
#include "context.h"
#include "list.h"
#include "message.h"
#include "proto.h"
#include "tlv.h"
#include "userstate.h"


otrl_instag_t chat_protocol_get_instag(OtrlUserState us, const
    OtrlMessageAppOps *ops, const char *accountname, const char *protocol)
{
    OtrlInsTag *ourInstanceTag;
    otrl_instag_t instag;

```

```
    ourInstanceTag = otrl_instag_find(us, accountname, protocol);
    if ((!ourInstanceTag) && ops->create_instag) {
        ops->create_instag(NULL, accountname, protocol);
        ourInstanceTag = otrl_instag_find(us, accountname, protocol);
    }

    if (ourInstanceTag && ourInstanceTag->instag >= OTRL_MIN_VALID_INSTAG)
    {
        instag = ourInstanceTag->instag;
    } else {
        instag = otrl_instag_get_new();
    }

    return instag;
}

int chat_protocol_app_info_refresh(const OtrlMessageAppOps *ops,
     ChatContextPtr ctx)
{
    OtrlChatInfoPtr info;

    info = chat_info_new_with_level(ctx);
    if(!info) { goto error; }

    ops->chat_info_refresh(NULL, info);

    chat_info_free(info);

    return 0;

error:
    return 1;
}

int chat_protocol_app_info_refresh_all(OtrlUserState us, const
     OtrlMessageAppOps *ops)
{
    OtrlListIteratorPtr iter;
    OtrlListNodePtr node;
    ChatContextPtr ctx;
    int err, ret = 0;

    iter = otrl_list_iterator_new(us->chat_context_list);
    if(!iter) { goto error; }
    while(otrl_list_iterator_has_next(iter)) {
        node = otrl_list_iterator_next(iter);
        ctx = otrl_list_node_get_payload(node);
        err = chat_protocol_app_info_refresh(ops, ctx);
        if(err) { ret = 1; }
    }

    return ret;

error:
    return 1;
}

int chat_protocol_is_in_use(OtrlUserState us, const char *accountname,
     const char *protocol, int *result)
{
    OtrlListIteratorPtr iter = NULL;
    OtrlListNodePtr node = NULL;
    ChatContextPtr ctx = NULL;
    ChatIdKeyPtr key = NULL;
    int res = 0;

    iter = otrl_list_iterator_new(us->chat_context_list);
    if(!iter) { goto error; }
    while(0 == res && otrl_list_iterator_has_next(iter)) {
        node = otrl_list_iterator_next(iter);
        ctx = otrl_list_node_get_payload(node);
        key = chat_context_get_identity_key(ctx);
```

```
        if(NULL != key) {
            if(0 == strcmp(accountname, chat_id_key_get_accountname(key))
    && 0 == strcmp(protocol, chat_id_key_get_protocol(key))) {
                res = 1;
            }
        }
    }
    otrl_list_iterator_free(iter);

    *result = res;
    return 0;

error:
    return 1;
}

int otrl_chat_protocol_id_key_read_file(OtrlUserState us, FILE *privf)
{
    OtrlListPtr key_list = NULL;

    key_list = us->chat_privkey_list;
    return chat_id_key_list_read_FILEp(key_list,
    &chat_dh_key_internalKeyOps, privf);
}

int otrl_chat_protocol_id_keys_write_file(OtrlUserState us, FILE *privf)
{
    OtrlListPtr key_list;

    key_list = us->chat_privkey_list;
    return chat_id_key_list_write_FILEp(key_list, privf);
}

int otrl_chat_protocol_id_key_generate_new(OtrlUserState us, const
    OtrlMessageAppOps *ops, const char *accountname, const char *protocol)
{
    int err = 0, in_use;

    fprintf(stderr, "libotr-mpOTR: otrl_chat_protocol_id_key_generate_new:
    start\n");

    err = chat_protocol_is_in_use(us, accountname, protocol, &in_use);
    if(err) { goto error; }

    if(!in_use) {
        err = chat_id_key_generate_new(us->chat_privkey_list, accountname,
    protocol, &chat_dh_key_internalKeyOps);
        if(err) { goto error; }
        ops->chat_privkeys_write(NULL);
    }

    fprintf(stderr, "libotr-mpOTR: otrl_chat_protocol_id_key_generate_new:
    end\n");

    return 0;

error:
    return 1;
}

OtrlListPtr otrl_chat_protocol_id_key_list_create(OtrlUserState us)
{
    OtrlListPtr list = NULL;

    list = chat_id_key_info_list_create(us->chat_privkey_list);
    if(!list) { goto error; }

    return list;

error:
    return NULL;
```

```
}

int chat_protocol_fingerprint_is_in_use(OtrlUserState us, OtrlChatFingerprintPtr fnprnt, int *result)
{
    OtrlListIteratorPtr iter1 = NULL, iter2 = NULL;
    ChatContextPtr ctx = NULL;
    ChatParticipantPtr part = NULL;
    OtrlChatFingerprintPtr cur_fnprnt = NULL;
    int res = 0;

    iter1 = otrl_list_iterator_new(us->chat_context_list);
    if(!iter1) { goto error; }
    while(0 == res && otrl_list_iterator_has_next(iter1))
    {
        ctx = otrl_list_node_get_payload(otrl_list_iterator_next(iter1));

        iter2 = otrl_list_iterator_new(chat_context_get_participants_list(ctx));
        if(!iter2) { goto error_with_iter1; }
        while(0 == res && otrl_list_iterator_has_next(iter2)) {
            part = otrl_list_node_get_payload(otrl_list_iterator_next(iter2));
            cur_fnprnt = chat_participant_get_fingerprint(part);
            if(cur_fnprnt == fnprnt) {
                res = 1;
            }
        }
        otrl_list_iterator_free(iter2);

    }
    otrl_list_iterator_free(iter1);

    *result = res;
    return 0;

error_with_iter1:
    otrl_list_iterator_free(iter1);
error:
    return 1;
}

int otrl_chat_protocol_fingerprints_read_file(OtrlUserState us, FILE *fingerfile)
{
    OtrlListPtr fingerlist;

    fingerlist = us->chat_fingerprints;
    return chat_fingerprint_read_FILEp(fingerlist, fingerfile);
}

int otrl_chat_protocol_fingerprints_write_file(OtrlUserState us, FILE *fingerfile)
{
    OtrlListPtr fingerlist;

    fingerlist = us->chat_fingerprints;
    return chat_fingerprint_write_FILEp(fingerlist, fingerfile);
}

void otrl_chat_protocol_fingerprint_verify(OtrlUserState us, const OtrlMessageAppOps *ops, OtrlChatFingerprintPtr fnprnt)
{
    chat_fingerprint_verify(fnprnt);
    ops->chat_fingerprints_write(NULL);

    chat_protocol_app_info_refresh_all(us, ops);
}

void otrl_chat_protocol_fingerprint_forget(OtrlUserState us, const OtrlMessageAppOps *ops, OtrlChatFingerprintPtr fnprnt)
{
```

```
    OtrlListPtr fingerlist;
    int err, inUse;

    err = chat_protocol_fingerprint_is_in_use(us, fnprnt, &inUse);
    if(err) { goto error; }

    if(0 == inUse) {
        fingerlist = us->chat_fingerprints;
        chat_fingerprint_forget(fingerlist, fnprnt);
        ops->chat_fingerprints_write(NULL);
    }

    chat_protocol_app_info_refresh_all(us, ops);

error:
    return;
}

int chat_protocol_participants_list_load_fingerprints(OtrlUserState us, ChatContextPtr ctx)
{
    OtrlListNodePtr node1, node2, node3;
    OtrlListIteratorPtr iter1, iter2;
    ChatParticipantPtr participant;
    OtrlChatFingerprintPtr fnprnt, newfinger;
    char *accountname = NULL, *protocol = NULL, *username = NULL;
    unsigned char *bytes = NULL;
    int trusted;

    iter1 = otrl_list_iterator_new(chat_context_get_participants_list(ctx));
    if(!iter1) { goto error; }

    while(otrl_list_iterator_has_next(iter1)) {
        node1 = otrl_list_iterator_next(iter1);
        participant = otrl_list_node_get_payload(node1);

        iter2 = otrl_list_iterator_new(us->chat_fingerprints);
        if(!iter2) { goto error_with_iter1; }

        while(otrl_list_iterator_has_next(iter2)) {
            node2 = otrl_list_iterator_next(iter2);
            fnprnt = otrl_list_node_get_payload(node2);

            accountname = otrl_chat_fingerprint_get_accountname(fnprnt);
            protocol = otrl_chat_fingerprint_get_protocol(fnprnt);
            username = otrl_chat_fingerprint_get_username(fnprnt);
            bytes = otrl_chat_fingerprint_get_bytes(fnprnt);
            trusted = otrl_chat_fingerprint_is_trusted(fnprnt);

            if(0 == strcmp(username, chat_participant_get_username(participant)) &&
                    0 == strcmp(accountname, chat_context_get_accountname(ctx)) &&
                    0 == strcmp(protocol, chat_context_get_protocol(ctx))) {

                newfinger = chat_fingerprint_new(accountname, protocol, username, bytes, trusted);
                if(!newfinger) { goto error_with_iter2; }

                node3 = otrl_list_insert(chat_participant_get_fingerprints(participant), newfinger);
                if(!node3) { goto error_with_newfinger; }
            }
        }
        otrl_list_iterator_free(iter2);

    }
    otrl_list_iterator_free(iter1);

```

```
    return 0;

error_with_newfinger:
    chat_fingerprint_free(newfinger);
error_with_iter2:
    otrl_list_iterator_free(iter2);
error_with_iter1:
    otrl_list_iterator_free(iter1);
error:
    return 1;
}

int chat_protocol_participants_list_init(OtrlUserState us, const OtrlMessageAppOps *ops, ChatContextPtr ctx)
{
    int err;
    char **usernames;
    unsigned int usernames_size;
    OtrlChatInfoPtr info;

    info = chat_info_new(ctx);
    if(!info) { goto error; }

    usernames = ops->chat_get_participants(NULL, info, &usernames_size);
    if(!usernames) { goto error_with_info; }

    err = chat_participant_list_from_usernames(chat_context_get_participants_list(ctx), usernames, usernames_size);
    if(err) { goto error_with_usernames; }

    err = chat_protocol_participants_list_load_fingerprints(us, ctx);
    if(err) { goto error_with_participants_list; }


    for(unsigned int i = 0; i < usernames_size; i++) { free(usernames[i]); }
    free(usernames);
    chat_info_free(info);

    return 0;

error_with_participants_list:
    otrl_list_clear(chat_context_get_participants_list(ctx));
error_with_usernames:
    for(unsigned int i = 0; i < usernames_size; i++) { free(usernames[i]); }
    free(usernames);
error_with_info:
    chat_info_free(info);
error:
    return 1;
}

void chat_protocol_reset(ChatContextPtr ctx)
{
    chat_context_reset(ctx);
}

int chat_protocol_add_sign(ChatContextPtr ctx, unsigned char **msg, size_t *msglen)
{
    Signature *aSign;
    unsigned char *sig = NULL, *buf;
    size_t siglen;
    int err;

    aSign = chat_sign_sign(chat_context_get_signing_key(ctx), *msg, *msglen);
    if(!aSign) { goto error; }

    err = chat_sign_signature_serialize(aSign, &sig, &siglen);
    if(err) { goto error_with_aSign; }
```

```

    buf = malloc((*msglen+siglen) * sizeof *buf);
    if(!buf) { goto error_with_sig; }

    memcpy(buf, *msg, *msglen);
    memcpy(&buf[*msglen], sig, siglen);

    free(*msg);
    free(sig);
    chat_sign_signature_free(aSign);

    *msg = buf;
    *msglen = *msglen+siglen;
    return 0;

error_with_sig:
    free(sig);
error_with_aSign:
    chat_sign_signature_free(aSign);
error:
    return 1;
}

int chat_protocol_send_message(const OtrlMessageAppOps *ops, ChatContextPtr ctx, ChatMessage *msg)
{
    OtrlChatInfoPtr info;
    char *message = NULL;
    unsigned char *buf = NULL;
    size_t buflen;
    int err;

    fprintf(stderr, "libotr-mpOTR: chat_protocol_send_message: start\n");

    buf = chat_message_serialize(msg, &buflen);
    if(!buf) { goto error; }

    if(chat_message_type_should_be_signed(msg->msgType) && CHAT_SIGNSTATE_SIGNED == chat_context_get_sign_state(ctx)) {
        err = chat_protocol_add_sign(ctx, &buf, &buflen);
        if(err) { goto error_with_buf; }
    }

    message = otrl_base64_otr_encode(buf, buflen);
    if(!message) { goto error_with_buf; }

    info = chat_info_new(ctx);
    if(!info) { goto error_with_message; }

    err = ops->chat_inject_message(NULL, info, message);
    if(err) { goto error_with_info; }

    chat_info_free(info);
    free(message);
    free(buf);

    fprintf(stderr, "libotr-mpOTR: chat_protocol_send_message: end\n");

    return 0;

error_with_info:
    chat_info_free(info);
error_with_message:
    free(message);
error_with_buf:
    free(buf);
error:
    return 1;

}

```

```
int chat_protocol_verify_sign(ChatContextPtr ctx, const char *sender,
    const unsigned char *msg, const size_t msglen) {
    Signature *sign;
    ChatParticipantPtr theSender;
    unsigned int their_pos;
    int err;

    err = chat_sign_signature_parse(
    &msg[msglen-CHAT_SIGN_SIGNATURE_LENGTH], &sign);
    if(err) { goto error; }

    theSender =
    chat_participant_find(chat_context_get_participants_list(ctx), sender,
    &their_pos);
    if(!theSender) { goto error_with_sign; }

    err = chat_sign_verify(chat_participant_get_sign_key(theSender), msg,
    msglen - CHAT_SIGN_SIGNATURE_LENGTH, sign);
    if(err) { goto error_with_sign; }

    chat_sign_signature_free(sign);

    return 0;

error_with_sign:
    chat_sign_signature_free(sign);
error:
    return 1;
}

int chat_protocol_pending_queue_add(ChatContextPtr ctx, const char
    *sender, unsigned char *msg, size_t msglen)
{
    ChatPendingPtr pending;
    OtrlListNodePtr node;

    pending = chat_pending_new(sender, msg, msglen);
    if(!pending) { goto error; }

    node = otrl_list_append(chat_context_get_pending_list(ctx), pending);
    if(!node) { goto error_with_pending; }

    return 0;

error_with_pending:
    chat_pending_free(pending);
error:
    return 1;
}

int chat_protocol_emit_event(const OtrlMessageAppOps *ops, const
    ChatContextPtr ctx, OtrlChatEventPtr event)
{
    OtrlChatInfoPtr info;

    info = chat_info_new(ctx);
    if(!info) { goto error; }

    ops->chat_handle_event(NULL, info, event);

    chat_info_free(info);

    return 0;

error:
    return 1;
}

int chat_protocol_emit_consensus_events(const OtrlMessageAppOps *ops,
    const ChatContextPtr ctx)
{
    OtrlChatEventPtr event;
```

```
    OtrlListIteratorPtr iter;
    OtrlListNodePtr cur;
    ChatParticipantPtr me, part;
    unsigned int pos;
    int err;

    me = chat_participant_find(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &pos);
    if(!me) { goto error; }

    iter = otrl_list_iterator_new(chat_context_get_participants_list(ctx));
    if(!iter) { goto error; }

    while(otrl_list_iterator_has_next(iter)) {
        cur = otrl_list_iterator_next(iter);
        part = otrl_list_node_get_payload(cur);

        if(part != me && 0 == chat_participant_get_consensus(part)) {
            event = chat_event_consensus_broken_new(
    chat_participant_get_username(part));
            if(!event) { goto error_with_iter; }

            err = chat_protocol_emit_event(ops, ctx, event);
            if(err) { goto error_with_event; }

            chat_event_free(event);
        }
    }

    otrl_list_iterator_free(iter);

    return 0;

error_with_event:
    chat_event_free(event);
error_with_iter:
    otrl_list_iterator_free(iter);
error:
    return 1;
}

int chat_protocol_emit_offer_received_event(const OtrlMessageAppOps *ops,
    const ChatContextPtr ctx, const char *username)
{
    OtrlChatEventPtr event;
    int err;

    event = chat_event_offer_received_new(username);
    if(!event) { goto error; }

    err = chat_protocol_emit_event(ops, ctx, event);
    if(err) { goto error_with_event; }

    chat_event_free(event);

    return 0;

error_with_event:
    chat_event_free(event);
error:
    return 1;
}

int chat_protocol_emit_starting_event(const OtrlMessageAppOps *ops, const
    ChatContextPtr ctx)
{
    OtrlChatEventPtr event;
    int err;

    event = chat_event_starting_new();
    if(!event) { goto error; }

```

```
    err = chat_protocol_emit_event(ops, ctx, event);
    if(err) { goto error_with_event; }

    chat_event_free(event);

    return 0;

error_with_event:
    chat_event_free(event);
error:
    return 1;
}

int chat_protocol_emit_started_event(const OtrlMessageAppOps *ops, const
    ChatContextPtr ctx)
{
    OtrlChatEventPtr event;
    int err;

    event = chat_event_started_new();
    if(!event) { goto error; }

    err = chat_protocol_emit_event(ops, ctx, event);
    if(err) { goto error_with_event; }

    chat_event_free(event);

    return 0;

error_with_event:
    chat_event_free(event);
error:
    return 1;
}

int chat_protocol_emit_unverified_participant_events(const
    OtrlMessageAppOps *ops, const ChatContextPtr ctx)
{
    OtrlChatEventPtr event;
    OtrlListIteratorPtr iter;
    OtrlListNodePtr cur;
    ChatParticipantPtr me, part;
    OtrlChatFingerprintPtr fnprnt;
    unsigned int pos;
    int err;

    me = chat_participant_find(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &pos);
    if(!me) { goto error; }

    iter = otrl_list_iterator_new(chat_context_get_participants_list(ctx));
    if(!iter) { goto error; }

    while(otrl_list_iterator_has_next(iter)) {
        cur = otrl_list_iterator_next(iter);
        part = otrl_list_node_get_payload(cur);

        if(part != me) {
            fnprnt = chat_participant_get_fingerprint(part);
            if(!fnprnt) { goto error_with_iter; }

            if(0 == otrl_chat_fingerprint_is_trusted(fnprnt)) {
                event = chat_event_unverified_participant_new(
    chat_participant_get_username(part));
                if(!event) { goto error_with_iter; }

                err = chat_protocol_emit_event(ops, ctx, event);
                if(err) { goto error_with_event; }

                chat_event_free(event);
            }
        }
```

```
    }

    otrl_list_iterator_free(iter);

    return 0;

error_with_event:
    chat_event_free(event);
error_with_iter:
    otrl_list_iterator_free(iter);
error:
    return 1;
}

int chat_protocol_emit_plaintext_received_event(const OtrlMessageAppOps
    *ops, const ChatContextPtr ctx, const char *sender, const char
    *message)
{
    OtrlChatEventPtr event;
    int err;

    event = chat_event_plaintext_received_new(sender, message);
    if(!event) { goto error; }

    err = chat_protocol_emit_event(ops, ctx, event);
    if(err) { goto error_with_event; }

    chat_event_free(event);

    return 0;

error_with_event:
    chat_event_free(event);
error:
    return 1;
}

int chat_protocol_emit_private_received_event(const OtrlMessageAppOps
    *ops, const ChatContextPtr ctx, const char *sender)
{
    OtrlChatEventPtr event;
    int err;

    event = chat_event_private_received_new(sender);
    if(!event) { goto error; }

    err = chat_protocol_emit_event(ops, ctx, event);
    if(err) { goto error_with_event; }

    chat_event_free(event);

    return 0;

error_with_event:
    chat_event_free(event);
error:
    return 1;
}

int chat_protocol_emit_finished_event(const OtrlMessageAppOps *ops, const
    ChatContextPtr ctx)
{
    OtrlChatEventPtr event;
    int err;

    event = chat_event_finished_new();
    if(!event) { goto error; }

    err = chat_protocol_emit_event(ops, ctx, event);
    if(err) { goto error_with_event; }

    chat_event_free(event);
```

```c

    return 0;

error_with_event:
    chat_event_free(event);
error:
    return 1;
}

int chat_protocol_handle_message(OtrlUserState us, const OtrlMessageAppOps *ops, ChatContextPtr ctx, const char *sender, const unsigned char *message, size_t messagelen, char **newmessagep, int *ignore, int *pending)
{
    int err, ignore_message = 0, ispending = 0, isrejected = 0;
    ChatMessageType type;
    ChatMessage *msg = NULL, *msgToSend = NULL;

    fprintf(stderr, "libotr-mpOTR: chat_protocol_handle_message: start\n");

    err = chat_message_parse_type(message, messagelen, &type);
    if(err) { goto error; }

    // Checking Session ID:
    // If the message contains a sid, we should check if it matches the sid of our context, otherwise reject it.
    // If we don't have obtained a sid for the session we add the message to the pending list.
    if(chat_message_type_contains_sid(type)) {
        ChatOfferInfoPtr offer_info = chat_context_get_offer_info(ctx);

        if(NULL == offer_info || CHAT_OFFERSTATE_FINISHED != chat_offer_info_get_state(offer_info)) {
            ispending = 1;
        } else {
            unsigned char *sid;
            err = chat_message_parse_sid(message, messagelen, &sid);
            if(err) { goto error; }

            if(memcmp(sid, chat_context_get_sid(ctx), CHAT_OFFER_SID_LENGTH)) {
                isrejected = 1;
            }

            free(sid);
        }
    }

    // Checking signature
    // If the singature verification fails, we reject the message
    // If we haven't entered the SINGED state yet, we add the message to the pending list
    if(!ispending && !isrejected ) {
        if(chat_message_type_should_be_signed(type)){
            if(CHAT_SIGNSTATE_SIGNED != chat_context_get_sign_state(ctx)) {
                ispending = 1;
            } else {
                err = chat_protocol_verify_sign(ctx, sender, message, messagelen);
                if(err) { goto error; }
                messagelen -= CHAT_SIGN_SIGNATURE_LENGTH;
            }
        }
    }

    // If sid and signature were verified, we try to handle the message based on its type
    if(!ispending && !isrejected) {
        msg = chat_message_parse(message, messagelen, sender);
        if(!msg) { goto error; }

        // CASE: Offer Message
```

```
    if(chat_offer_is_my_message(msg)) {

        // If we haven't done that yet, initialize the participant list and the offer info
        if(NULL == chat_context_get_offer_info(ctx)) {

            // Library-Application Communication
            chat_protocol_emit_offer_received_event(ops, ctx, sender);

            err = chat_protocol_participants_list_init(us, ops, ctx);
            if(err) { goto error_with_msg; }

            err = chat_offer_info_init(ctx, otrl_list_size(chat_context_get_participants_list(ctx)));
            if(err) { goto error_with_msg; }
        }

        ChatOfferInfoPtr offer_info = chat_context_get_offer_info(ctx);

        if(CHAT_OFFERSTATE_FINISHED == chat_offer_info_get_state(offer_info)) {
            //reject
        } else {

            err = chat_offer_handle_message(ctx, msg, &msgToSend);
            if(err) { goto error_with_msg; }
            if(msgToSend) {
                err = chat_protocol_send_message(ops, ctx, msgToSend);
                if(err) { goto error_with_msgToSend; }
                chat_message_free(msgToSend);
                msgToSend = NULL;
            }

            if(NULL != offer_info && CHAT_OFFERSTATE_FINISHED == chat_offer_info_get_state(offer_info)) {
                // Load or generate our private key
                ChatIdKeyPtr id_key = NULL;
                err = chat_id_key_find(us->chat_privkey_list, chat_context_get_accountname(ctx), chat_context_get_protocol(ctx), &id_key);
                if(err) { goto error_with_msg; }

                if(!id_key) {
                    ops->chat_privkey_create(NULL, chat_context_get_accountname(ctx), chat_context_get_protocol(ctx));
                    err = chat_id_key_find(us->chat_privkey_list, chat_context_get_accountname(ctx), chat_context_get_protocol(ctx), &id_key);
                    if(err || NULL == id_key) { goto error_with_msg; }
                }
                chat_context_set_identity_key(ctx, id_key);

                // Initiate dske
                err = chat_dske_init(ctx, &msgToSend);
                if(err) { goto error_with_msg; }
                err = chat_protocol_send_message(ops, ctx, msgToSend);
                if(err) { goto error_with_msgToSend; }
                chat_message_free(msgToSend);
                msgToSend = NULL;
            }
        }
        ignore_message = 1;

    // CASE: DSKE Message
    } else if(chat_dske_is_my_message(msg)) {
        ChatOfferInfoPtr offer_info = chat_context_get_offer_info(ctx);
        ChatDSKEInfoPtr dske_info = chat_context_get_dske_info(ctx);
        ChatGKAInfoPtr gka_info = chat_context_get_gka_info(ctx);

        if(NULL == dske_info || NULL == offer_info || chat_offer_info_get_state(offer_info) != CHAT_OFFERSTATE_FINISHED) {
            ispending = 1;
```

```
        } else if(CHAT_DSKESTATE_FINISHED == chat_dske_info_get_state(dske_info)) {
            // reject
        } else {
            err = chat_dske_handle_message(ctx, msg, us->chat_fingerprints, &msgToSend);
            if(err) { goto error_with_msg; }

            if(msgToSend) {
                err = chat_protocol_send_message(ops, ctx, msgToSend);
                if(err) { goto error_with_msgToSend; }
                chat_message_free(msgToSend);
                msgToSend = NULL;
            }

            if(NULL != dske_info && CHAT_DSKESTATE_FINISHED == chat_dske_info_get_state(dske_info) &&
                    (NULL == gka_info || CHAT_GKASTATE_NONE == chat_gka_info_get_state(gka_info))) {

                chat_context_set_sign_state(ctx, CHAT_SIGNSTATE_SIGNED);

                err = chat_gka_init(ctx, &msgToSend);
                if(err) { goto error_with_msg; }

                if(msgToSend) {
                    err = chat_protocol_send_message(ops, ctx, msgToSend);
                    if(err) { goto error_with_msgToSend; }
                    chat_message_free(msgToSend);
                    msgToSend = NULL;
                }
            }
        }
        ignore_message = 1;

    // CASE: GKA Message
    } else if(chat_gka_is_my_message(msg)) {
        ChatDSKEInfoPtr dske_info = chat_context_get_dske_info(ctx);
        ChatGKAInfoPtr gka_info = chat_context_get_gka_info(ctx);

        if(NULL == dske_info || CHAT_DSKESTATE_FINISHED != chat_dske_info_get_state(dske_info)) {
            ispending = 1;
        } else if(CHAT_GKASTATE_FINISHED == chat_gka_info_get_state(gka_info)) {
            // reject
        } else {
            err = chat_gka_handle_message(ctx, msg, &msgToSend);
            if(err) { goto error_with_msg; }

            if(msgToSend) {
                err = chat_protocol_send_message(ops, ctx, msgToSend);
                if(err) { goto error_with_msgToSend; }
                chat_message_free(msgToSend);
                msgToSend = NULL;
            }

            if(CHAT_GKASTATE_FINISHED == chat_gka_info_get_state(gka_info)) {
                err = chat_attest_init(ctx, &msgToSend);
                if(err) { goto error_with_msg; }
                if(msgToSend) {
                    err = chat_protocol_send_message(ops, ctx, msgToSend);
                    if(err) { goto error_with_msgToSend; }
                    chat_message_free(msgToSend);
                    msgToSend = NULL;
                }
            }
        }
```

```
        ignore_message = 1;

    // CASE: Attest Message
    } else if(chat_attest_is_my_message(msg)) {

        ChatGKAInfoPtr gka_info = chat_context_get_gka_info(ctx);
        ChatAttestInfoPtr attest_info = chat_context_get_attest_info(ctx);

        if(NULL == attest_info || CHAT_GKASTATE_FINISHED != chat_gka_info_get_state(gka_info)) {
            ispending = 1;
        } else if(CHAT_ATTESTSTATE_FINISHED == chat_attest_info_get_state(attest_info)) {
            // reject
        } else {
            err = chat_attest_handle_message(ctx, msg, &msgToSend);
            if(err) { goto error_with_msg; }

            if(msgToSend) {
                err = chat_protocol_send_message(ops, ctx, msgToSend);
                if(err) { goto error_with_msgToSend; }
                chat_message_free(msgToSend);
                msgToSend = NULL;
            }

            if(NULL != attest_info && CHAT_ATTESTSTATE_FINISHED == chat_attest_info_get_state(attest_info)) {
                err = chat_shutdown_init(ctx);
                if(err) { goto error_with_msg; }

                // Library-Application Communication
                chat_protocol_emit_started_event(ops, ctx);
                chat_protocol_emit_unverified_participant_events(ops, ctx);
                chat_protocol_app_info_refresh(ops, ctx);
            }
        }
        ignore_message = 1;

    // CASE: Communication Message
    } else if(chat_communication_is_my_message(msg)) {

        ChatAttestInfoPtr attest_info = chat_context_get_attest_info(ctx);

        //pending case
        if(NULL != attest_info && CHAT_ATTESTSTATE_AWAITING == chat_attest_info_get_state(attest_info)) {
            ispending = 1;

        // rejecting
        } else if(OTRL_MSGSTATE_PLAINTEXT == chat_context_get_msg_state(ctx) || OTRL_MSGSTATE_FINISHED == chat_context_get_msg_state(ctx)) {
            chat_protocol_emit_private_received_event(ops, ctx, sender);

        // hnadling
        } else {
            char *plaintext;
            err = chat_communication_handle_msg(ctx, msg, NULL, &plaintext);
            if(err) { goto error_with_msg; }
            *newmessagep = plaintext;
        }

    // CASE: Shutdown Message
    } else if (chat_shutdown_is_my_message(msg)) {

```

```
        ChatAttestInfoPtr attest_info = chat_context_get_attest_info(ctx);

        if(NULL == attest_info || CHAT_ATTESTSTATE_FINISHED != chat_attest_info_get_state(attest_info)) {
            // reject
        } else {
            ChatShutdownInfoPtr shutdown_info = chat_context_get_shutdown_info(ctx);
            ChatShutdownState prevState = chat_shutdown_info_get_state(shutdown_info);

            err = chat_shutdown_handle_message(ctx, msg, &msgToSend);
            if(err) { goto error_with_msg; }

            if(msgToSend) {
                err = chat_protocol_send_message(ops, ctx, msgToSend);
                if(err) { goto error_with_msgToSend; }
                chat_message_free(msgToSend);
                msgToSend = NULL;
            }

            if(CHAT_SHUTDOWNSTATE_AWAITING_DIGESTS == chat_shutdown_info_get_state(shutdown_info) && CHAT_SHUTDOWNSTATE_AWAITING_SHUTDOWNS == prevState) {
                err = chat_shutdown_send_digest(ctx, &msgToSend);
                if(err) { goto error_with_msgToSend;}

                err = chat_protocol_send_message(ops, ctx, msgToSend);
                if(err) { goto error_with_msgToSend; }

                chat_message_free(msgToSend);
                msgToSend = NULL;
            }

            if(CHAT_SHUTDOWNSTATE_AWAITING_ENDS == chat_shutdown_info_get_state(shutdown_info) && CHAT_SHUTDOWNSTATE_AWAITING_DIGESTS == prevState) {
                err = chat_shutdown_send_end(ctx, &msgToSend);
                if(err) { goto error_with_msgToSend;}

                err = chat_protocol_send_message(ops, ctx, msgToSend);
                if(err) { goto error_with_msgToSend; }

                chat_message_free(msgToSend);
                msgToSend = NULL;
            }

            if(CHAT_SHUTDOWNSTATE_FINISHED == chat_shutdown_info_get_state(shutdown_info)) {
                chat_context_set_msg_state(ctx, OTRL_MSGSTATE_FINISHED);

                err = chat_shutdown_release_secrets(ctx, &msgToSend);
                if(err) { goto error_with_msgToSend;}

                err = chat_protocol_send_message(ops, ctx, msgToSend);
                if(err) { goto error_with_msgToSend; }
                chat_message_free(msgToSend);
                msgToSend = NULL;

                // Library-Application Communication
                chat_protocol_emit_finished_event(ops, ctx);
                chat_protocol_emit_consensus_events(ops, ctx);
                chat_protocol_app_info_refresh(ops, ctx);

                chat_protocol_reset(ctx);
            }
        }

        ignore_message = 1;
    }
```

```c
    }

    *pending = ispending;
    *ignore = ignore_message;

    fprintf(stderr, "libotr-mpOTR: chat_protocol_handle_message: end\n");
    return 0;

error_with_msgToSend:
    chat_message_free(msg);
error_with_msg:
    chat_message_free(msg);
error:
    return 1;
}

int chat_protocol_handle_pending(OtrlUserState us, const OtrlMessageAppOps *ops, ChatContextPtr ctx) {
    OtrlListIteratorPtr iter;
    OtrlListNodePtr cur;
    ChatPendingPtr pending;
    unsigned short int flag = 1;
    int err, ispending, ignore;

    fprintf(stderr, "libotr-mpOTR: chat_protocol_handle_pending: start\n");

    if(otrl_list_size(chat_context_get_pending_list(ctx)) > 0) {
        fprintf(stderr, "=========================================================\n");
        fprintf(stderr, "libotr-mpOTR: chat_protocol_handle_pending: PENDING LIST:\n");
        otrl_list_dump(chat_context_get_pending_list(ctx));
        fprintf(stderr, "=========================================================\n");
    }

    while(flag) {
        flag = 0;

        iter = otrl_list_iterator_new(chat_context_get_pending_list(ctx));
        if(!iter) { goto error; }

        while(otrl_list_iterator_has_next(iter)) {
            cur = otrl_list_iterator_next(iter);
            pending = otrl_list_node_get_payload(cur);

            err = chat_protocol_handle_message(us, ops, ctx, chat_pending_get_sender(pending), chat_pending_get_msg(pending), chat_pending_get_msglen(pending), NULL, &ignore, &ispending);
            if(err) { goto error_with_iter; }

            if(!ispending) {
                otrl_list_remove_and_free(chat_context_get_pending_list(ctx), cur);
                flag = 1;
            }
        }

        otrl_list_iterator_free(iter);
    }

    fprintf(stderr, "libotr-mpOTR: chat_protocol_handle_pending: end\n");

    return 0;

error_with_iter:
    otrl_list_iterator_free(iter);
error:
    return 1;
}

```

```
int otrl_chat_protocol_receiving(OtrlUserState us, const OtrlMessageAppOps *ops,
    void *opdata, const char *accountname, const char *protocol,
    const char *sender, otrl_chat_token_t chat_token, const char *message,
    char **newmessagep, OtrlTLV **tlvsp)
{
    ChatContextPtr ctx;
    otrl_instag_t instag;
    int ignore_message = 0; // flag to determine if the message should be ignored
    int ispending, err;
    unsigned char *buf;
    size_t buflen;

    fprintf(stderr, "libotr-mpOTR: otrl_chat_protocol_receiving: start\n");

    if( !accountname || !protocol || !sender || !message || !newmessagep) { goto error; }

    instag = chat_protocol_get_instag(us, ops, accountname, protocol);

    ctx = chat_context_find_or_add(us->chat_context_list, accountname, protocol, chat_token, instag);
    if(!ctx) { goto error; }

    if(!chat_message_is_otr(message)) {

        if (OTRL_MSGSTATE_PLAINTEXT != chat_context_get_msg_state(ctx)) {
            chat_protocol_emit_plaintext_received_event(ops, ctx, sender, message);
            ignore_message = 1;
        }

    } else {

        err = otrl_base64_otr_decode(message, &buf, &buflen);
        if(err) { goto error; }

        err = chat_protocol_handle_message(us, ops, ctx, sender, buf, buflen, newmessagep, &ignore_message, &ispending);
        if(err) { goto error_with_buf; }

        if(ispending) {
            err = chat_protocol_pending_queue_add(ctx, sender, buf, buflen);
            if(err) { goto error_with_buf; }
        } else {
            err = chat_protocol_handle_pending(us, ops, ctx);
            if(err) { goto error_with_buf; }
        }

        free(buf);
    }

    fprintf(stderr, "libotr-mpOTR: otrl_chat_protocol_receiving: end\n");
    return ignore_message;

error_with_buf:
    free(buf);
error:
    return 1;
}

int otrl_chat_protocol_sending(OtrlUserState us,
    const OtrlMessageAppOps *ops,
    void *opdata, const char *accountname, const char *protocol,
    const char *message, otrl_chat_token_t chat_token, OtrlTLV *tlvs,
    char **messagep, OtrlFragmentPolicy fragPolicy)
{
    ChatContextPtr ctx;
    otrl_instag_t instag;
    unsigned char *buf;
```

```
    ChatMessage *msg;
    size_t buflen;
    int err;

    fprintf(stderr, "libotr-mpOTR: otrl_chat_message_sending: start\n");

    if( !accountname || !protocol || !message) { goto error; }

    instag = chat_protocol_get_instag(us, ops, accountname, protocol);

    ctx = chat_context_find_or_add(us->chat_context_list, accountname, protocol, chat_token, instag);
    if(!ctx) { goto error; }

    switch(chat_context_get_msg_state(ctx)) {
        case OTRL_MSGSTATE_PLAINTEXT:
            fprintf(stderr, "libotr-mpOTR: otrl_chat_message_sending: case OTRL_MSGSTATE_PLAINTEXT\n");
            break;
        case OTRL_MSGSTATE_ENCRYPTED:
            fprintf(stderr, "libotr-mpOTR: otrl_chat_message_sending: case OTRL_MSGSTATE_ENCRYPTED\n");

            err = chat_communication_broadcast(ctx, message, &msg);
            if(err) { goto error; }

            buf = chat_message_serialize(msg, &buflen);
            if(!buf) { goto error_with_msg; }

            if(chat_message_type_should_be_signed(msg->msgType) && CHAT_SIGNSTATE_SIGNED == chat_context_get_sign_state(ctx) ) {
                err = chat_protocol_add_sign(ctx, &buf, &buflen);
                if(err) { goto error_with_buf; }
            }

            *messagep = otrl_base64_otr_encode(buf, buflen);
            if(!*messagep) { goto error_with_buf; }

            free(buf);

            chat_message_free(msg);

            break;
        case OTRL_MSGSTATE_FINISHED:
            fprintf(stderr, "libotr-mpOTR: otrl_chat_message_sending: case OTRL_MSGSTATE_FINISHED\n");
            break;
    }

    fprintf(stderr, "libotr-mpOTR: otrl_chat_message_sending: end\n");
    return 0;

error_with_buf:
    free(buf);
error_with_msg:
    chat_message_free(msg);
error:
    return 1;
}

int otrl_chat_protocol_send_query(OtrlUserState us,
        const OtrlMessageAppOps *ops,
        const char *accountname, const char *protocol,
        otrl_chat_token_t chat_token, OtrlFragmentPolicy fragPolicy)
{
    ChatMessage *msgToSend;
    ChatContextPtr ctx;
    otrl_instag_t instag;
    int err;

    fprintf(stderr, "libotr-mpOTR: otrl_chat_protocol_send_query: start\n");
```

```
    instag = chat_protocol_get_instag(us, ops, accountname, protocol);

    ctx = chat_context_find_or_add(us->chat_context_list, accountname, protocol, chat_token, instag);
    if(!ctx) { goto error; }

    err = chat_protocol_participants_list_init(us, ops, ctx);
    if(err) { goto error; }

    err = chat_offer_info_init(ctx, otrl_list_size(chat_context_get_participants_list(ctx)));
    if(err) { goto error_with_msg; }

    err = chat_offer_start(ctx, &msgToSend);
    if(err) { goto error; }

    chat_protocol_emit_starting_event(ops, ctx);

    err = chat_protocol_send_message(ops, ctx, msgToSend);
    if(err) { goto error_with_msg; }

    chat_message_free(msgToSend);

    fprintf(stderr, "libotr-mpOTR: otrl_chat_protocol_send_query: end\n");

    return 0;

error_with_msg:
    chat_message_free(msgToSend);
error:
    return 1;
}

int otrl_chat_protocol_shutdown(OtrlUserState us, const OtrlMessageAppOps *ops,
        const char *accountname, const char *protocol, otrl_chat_token_t chat_token)
{
    ChatMessage *msgToSend;
    ChatContextPtr ctx;
    ChatShutdownInfoPtr shutdown_info;
    OtrlListPtr context_list;
    otrl_instag_t instag;
    int err;

    fprintf(stderr, "libotr-mpOTR: otrl_chat_protocol_shutdown: start\n");

    instag = chat_protocol_get_instag(us, ops, accountname, protocol);
    context_list = us->chat_context_list;

    ctx = chat_context_find(context_list, accountname, protocol, chat_token, instag);
    if(!ctx) { goto error; }

    shutdown_info = chat_context_get_shutdown_info(ctx);

    if(NULL == shutdown_info || CHAT_SHUTDOWNSTATE_AWAITING_SHUTDOWNS != chat_shutdown_info_get_state(shutdown_info)) { goto error; }

    err = chat_shutdown_send_shutdown(ctx, &msgToSend);
    if(err) { goto error; }

    err = chat_protocol_send_message(ops, ctx, msgToSend);
    if(err) { goto error_with_msg; }

    chat_message_free(msgToSend);

    fprintf(stderr, "libotr-mpOTR: otrl_chat_protocol_showtdown: end\n");
    return 0;

error_with_msg:
```

```
    chat_message_free(msgToSend);
error:
    return 1;
}
```

Listing A′.2: chat_protocol.c

Appendix B′

Offer source code

```c
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 */

#ifndef CHAT_OFFER_H_
#define CHAT_OFFER_H_

#include "message.h"
#include "chat_types.h"

typedef enum {
    CHAT_OFFERSTATE_NONE,
    CHAT_OFFERSTATE_AWAITING,
    CHAT_OFFERSTATE_FINISHED
} ChatOfferState;

ChatOfferInfoPtr chat_offer_info_new(size_t size);

void chat_offer_info_free(ChatOfferInfoPtr info);

ChatOfferState chat_offer_info_get_state(ChatOfferInfoPtr offer_info);

int chat_offer_info_init(ChatContextPtr ctx, size_t size);

int chat_offer_handle_message(ChatContextPtr ctx, const ChatMessage *msg, ChatMessage **msgToSend);

int chat_offer_start(ChatContextPtr ctx, ChatMessage **msgToSend);

int chat_offer_is_my_message(const ChatMessage *msg);

#endif /* CHAT_OFFER_H_ */
```

Listing B′.1: chat_offer.h

```c
/*
 *  Off-the-Record Messaging library
```

```
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 */

#include "chat_offer.h"

#include <stdlib.h>
#include <string.h>
#include <gcrypt.h>

#include "chat_context.h"
#include "chat_message.h"
#include "chat_participant.h"
#include "chat_types.h"

struct ChatOfferInfo {
        size_t size;
        size_t added;
        unsigned char **sid_contributions;
        ChatOfferState state;
};

unsigned char * chat_offer_compute_sid(unsigned char **sid_contributions, size_t size)
{
    unsigned int i;
    unsigned char *sid;
    gcry_md_hd_t md;
    gcry_error_t err;

    fprintf(stderr, "libotr-mpOTR: chat_offer_compute_sid: start\n");

    err = gcry_md_open(&md, GCRY_MD_SHA512, 0);
    if(err) { goto error; }

    for(i=0; i<size; i++) {
        if(sid_contributions[i] == NULL) { goto error; }
        gcry_md_write(md, sid_contributions[i], CHAT_OFFER_SID_CONTRIBUTION_LENGTH);
    }

    sid = malloc(CHAT_OFFER_SID_LENGTH * sizeof *sid);
    if(sid == NULL) { goto error_with_md; }

    memcpy(sid, gcry_md_read(md, GCRY_MD_SHA512), CHAT_OFFER_SID_LENGTH);
    gcry_md_close(md);

    fprintf(stderr, "libotr-mpOTR: chat_offer_compute_sid: computed sid: ");
    for(size_t i = 0; i < CHAT_OFFER_SID_LENGTH; i++) fprintf(stderr,"%02X", sid[i]);
    fprintf(stderr,"\n");

    fprintf(stderr, "libotr-mpOTR: chat_offer_compute_sid: emd\n");

    return sid;
```

```c

error_with_md:
    gcry_md_close(md);
error:
    return NULL;
}

unsigned char * chat_offer_create_sid_contribution()
{
    unsigned char *rand_bytes, *sid_contribution = NULL;

    sid_contribution = malloc(CHAT_OFFER_SID_CONTRIBUTION_LENGTH * sizeof *sid_contribution);
    if(!sid_contribution) { goto error; }
    rand_bytes = gcry_random_bytes(CHAT_OFFER_SID_CONTRIBUTION_LENGTH, GCRY_STRONG_RANDOM);
    memcpy(sid_contribution, rand_bytes, CHAT_OFFER_SID_CONTRIBUTION_LENGTH);
    gcry_free(rand_bytes);

    return sid_contribution;

error:
    return NULL;
}

ChatOfferInfoPtr chat_offer_info_new(size_t size)
{
    ChatOfferInfoPtr offer_info;

    offer_info = malloc(sizeof *offer_info);
    if(!offer_info) { goto error; }

    offer_info->size = size;
    offer_info->added = 0;
    offer_info->sid_contributions = calloc(size, sizeof *offer_info->sid_contributions);
    if(!offer_info->sid_contributions) { goto err_with_offer_info; }

    offer_info->state = CHAT_OFFERSTATE_NONE;

    return offer_info;

err_with_offer_info:
    free(offer_info);
error:
    return NULL;
}

void chat_offer_info_free(ChatOfferInfoPtr info) {
    unsigned int i;

    if(info) {
        for(i=0; i<info->size; i++) {
            free(info->sid_contributions[i]);
        }
    }
    free(info);
}

ChatOfferState chat_offer_info_get_state(ChatOfferInfoPtr offer_info)
{
    return offer_info->state;
}

int chat_offer_info_init(ChatContextPtr ctx, size_t size) {
    ChatOfferInfoPtr offer_info;

    offer_info = chat_offer_info_new(size);
    if(!offer_info) { goto error; }

    chat_context_set_offer_info(ctx, offer_info);
```

```

    return 0;

error:
    return 1;
}

int chat_offer_add_sid_contribution(ChatOfferInfoPtr offer_info, const unsigned char *sid_contribution, unsigned int position)
{
    unsigned char *contribution;

    if(position >= offer_info->size) { goto error; }
    if(offer_info->sid_contributions[position] != NULL) { goto error; }

    contribution = malloc(CHAT_OFFER_SID_CONTRIBUTION_LENGTH * sizeof *contribution);
    if(!contribution) { goto error; }
    memcpy(contribution, sid_contribution, CHAT_OFFER_SID_CONTRIBUTION_LENGTH);

    offer_info->sid_contributions[position] = contribution;
    offer_info->added++;

    return 0;

error:
    return 1;
}

int chat_offer_sid_contribution_exists(ChatOfferInfoPtr offer_info, unsigned int position)
{
    if(offer_info->sid_contributions[position] == NULL) {
        return 0;
    } else {
        return 1;
    }
}

int chat_offer_is_ready(ChatOfferInfoPtr offer_info)
{
    if(offer_info->added < offer_info->size) {
        return 0;
    } else {
        return 1;
    }
}

int chat_offer_handle_message(ChatContextPtr ctx, const ChatMessage *msg, ChatMessage **msgToSend)
{
    int err;
    unsigned int their_pos, our_pos;
    unsigned char *our_contribution, *sid;
    ChatOfferInfoPtr offer_info;
    ChatMessage *newmsg = NULL;
    ChatMessagePayloadOffer *payload = msg->payload;

    *msgToSend = NULL;

    fprintf(stderr, "libotr-mpOTR: chat_offer_handle_message: start\n");

    offer_info = chat_context_get_offer_info(ctx);
    if(!offer_info) { goto error; }

    err = chat_participant_get_position(chat_context_get_participants_list(ctx), msg->senderName, &their_pos);
    if(err) { goto error; }
    if(their_pos != payload->position || their_pos >= offer_info->size) {
        goto error;
```

```
    }

    if( chat_offer_sid_contribution_exists(offer_info, their_pos)) {
        goto error;
    }

    err = chat_offer_add_sid_contribution(offer_info,
    payload->sid_contribution, their_pos);
    if(err) { goto error; }

    err =
    chat_participant_get_position(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &our_pos);
    if(err) { goto error; }

    if(!chat_offer_sid_contribution_exists(offer_info, our_pos)) {
        our_contribution = chat_offer_create_sid_contribution();
        if(!our_contribution) { goto error; }

        err = chat_offer_add_sid_contribution(offer_info,
    our_contribution, our_pos);
        if(err) { free(our_contribution); goto error; }

        newmsg = chat_message_offer_new(ctx, our_contribution, our_pos);

        if(!newmsg) { goto error; }

        free(our_contribution);
    }

    if(chat_offer_is_ready(offer_info)) {
        sid = chat_offer_compute_sid(offer_info->sid_contributions,
    offer_info->size);
        if(!sid) { goto error; }
        memcpy(chat_context_get_sid(ctx), sid, CHAT_OFFER_SID_LENGTH);
        free(sid);
        offer_info->state = CHAT_OFFERSTATE_FINISHED;
    }

    *msgToSend = newmsg;

    fprintf(stderr, "libotr-mpOTR: chat_offer_handle_message: end\n");

    return 0;

error:
    return 1;
}

int chat_offer_start(ChatContextPtr ctx, ChatMessage **msgToSend)
{
    ChatOfferInfoPtr offer_info;
    unsigned int our_pos;
    unsigned char *our_contribution, *sid;
    ChatMessage *newmsg = NULL;
    int err;

    fprintf(stderr, "libotr-mpOTR: chat_offer_start: start\n");

    offer_info = chat_context_get_offer_info(ctx);
    if(!offer_info) { goto error; }

    *msgToSend = NULL;

    err =
    chat_participant_get_position(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &our_pos);
    if(err) { goto error; }

    our_contribution = chat_offer_create_sid_contribution();
    if(!our_contribution) { goto error; }

```

```
    newmsg = chat_message_offer_new(ctx, our_contribution, our_pos);
    if(!newmsg) { goto error_with_our_contribution; }

    err = chat_offer_add_sid_contribution(offer_info, our_contribution,
    our_pos);
    if(err) { goto erro_with_newmsg; }

    if(chat_offer_is_ready(offer_info)) {
        sid = chat_offer_compute_sid(offer_info->sid_contributions,
    offer_info->size);
        if(!sid) { goto error; }

        memcpy(chat_context_get_sid(ctx), sid, CHAT_OFFER_SID_LENGTH);
        offer_info->state = CHAT_OFFERSTATE_FINISHED;
    }

    *msgToSend = newmsg;

    free(our_contribution);

    fprintf(stderr, "libotr-mpOTR: chat_offer_start: end\n");

    return 0;

erro_with_newmsg:
    chat_message_free(newmsg);
error_with_our_contribution:
    free(our_contribution);
error:
    return 1;
}

int chat_offer_is_my_message(const ChatMessage *msg)
{
    ChatMessageType msg_type = msg->msgType;

    switch(msg_type) {
        case CHAT_MSGTYPE_OFFER:
            return 1;
        default:
            return 0;
    }
}
```

Listing B′.2: chat_offer.c

Appendix Γ′

DSKE source code

```
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros
    <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos
    <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
    02110-1301 USA
 */

#include "chat_types.h"

typedef enum {
    CHAT_DSKESTATE_NONE,
    CHAT_DSKESTATE_AWAITING_KEYS,
    CHAT_DSKESTATE_FINISHED
} ChatDSKEState;

int chat_dske_init(ChatContextPtr ctx, ChatMessage **msgToSend);

void chat_dske_info_free(ChatDSKEInfoPtr dske_info);

ChatDSKEState chat_dske_info_get_state(ChatDSKEInfoPtr dske_info);

int chat_dske_is_my_message(const ChatMessage *msg);

int chat_dske_handle_message(ChatContextPtr ctx, ChatMessage *msg,
        OtrlListPtr fnprnt_list, ChatMessage **msgToSend);
```

Listing Γ′.1: chat_dske.h

```
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros
    <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos
    <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```c
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
    02110-1301 USA
 */

#include "chat_dske.h"

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "chat_context.h"
#include "chat_dake.h"
#include "chat_fingerprint.h"
#include "chat_message.h"
#include "chat_participant.h"
#include "chat_sign.h"
#include "chat_types.h"
#include "list.h"

struct ChatDSKEInfo {
    ChatDSKEState state;
    DAKEInfo dake_info;
    unsigned int remaining;
};

void chat_dske_info_free(ChatDSKEInfoPtr dske_info)
{
    if(dske_info) {
        chat_dake_destroy_info(&dske_info->dake_info);
    }
    free(dske_info);
}

ChatDSKEState chat_dske_info_get_state(ChatDSKEInfoPtr dske_info)
{
    return dske_info->state;
}

int chat_dske_init(ChatContextPtr ctx, ChatMessage **msgToSend)
{
    ChatDSKEInfoPtr dske_info;
    unsigned int my_pos;
    ChatParticipantPtr me, participant;
    SignKey *sign_key, *sign_key_pub_copy;
    DAKE *dake;
    OtrlListIteratorPtr iter, iter2;
    OtrlListNodePtr cur;
    DAKE_handshake_message_data *dataToSend;
    ChatMessage *msg = NULL;
    int err;

    fprintf(stderr,"chat_dske_init: start\n");

    /* Allocate memory for the info struct */
    dske_info = malloc(sizeof *dske_info);
    if(!dske_info) { goto error; }

    /* Find us in the participant list */
    me = chat_participant_find(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &my_pos);
    if(!me) { goto error_with_dske_info; }

    /* Get what values we should broadcast to every other user. dataToSend
    will
     * contain the data that will be sent in the handshake message. */
```

```
    err = chat_dake_init_keys(&dske_info->dake_info, chat_context_get_identity_key(ctx), chat_context_get_accountname(ctx),
                               chat_context_get_protocol(ctx), &dataToSend);
    if(err) { goto error_with_dske_info; }

    /* Initiate a dake for each participant. The dake struct holds information
     * regarding each individual DAKE with each participant. */
    iter = otrl_list_iterator_new(chat_context_get_participants_list(ctx));

    while(otrl_list_iterator_has_next(iter)) {
        cur = otrl_list_iterator_next(iter);
        participant = otrl_list_node_get_payload(cur);

        dake = malloc(sizeof *dake);
        if(!dake) { goto error_with_participants; }

        chat_participant_set_dake(participant, dake);

        err = chat_dake_init(chat_participant_get_dake(participant), &dske_info->dake_info);
        if(err) { goto error_with_participants; }
    }

    /* Create the message we should send */
    msg = chat_message_dake_handshake_new(ctx, dataToSend);
    if(!msg) { goto error_with_participants; }

    /* Change the protocol state */
    dske_info->remaining = otrl_list_size(chat_context_get_participants_list(ctx)) - 1;
    dske_info->state = CHAT_DSKESTATE_AWAITING_KEYS;

    fprintf(stderr,"chat_dske_init: before genkey\n");

    /* Generate an ephemeral signing key for this session */
    sign_key = chat_sign_genkey();
    if(!sign_key) { goto error_with_msg; }

    /* Copy the public part of the signing key in me */
    sign_key_pub_copy = chat_sign_copy_pub(sign_key);
    if(!sign_key) { goto error_with_sign_key; }

    chat_context_set_signing_key(ctx, sign_key);
    chat_participant_set_sign_key(me, sign_key_pub_copy);

    chat_context_set_dske_info(ctx, dske_info);

    *msgToSend = msg;

    fprintf(stderr,"chat_dske_init: end\n");
    return 0;

error_with_sign_key:
    chat_sign_destroy_key(sign_key);
error_with_msg:
    chat_message_free(msg);
error_with_participants:
    iter2 = otrl_list_iterator_new(chat_context_get_participants_list(ctx));
    if(iter2) {
        while(otrl_list_iterator_has_next(iter2)) {
            cur = otrl_list_iterator_next(iter2);
            participant = otrl_list_node_get_payload(cur);
            if(NULL != chat_participant_get_dake(participant)){
                chat_dake_destroy(chat_participant_get_dake(participant));
                free(chat_participant_get_dake(participant));
                chat_participant_set_dake(participant, NULL);
            }
        }
    }
    otrl_list_iterator_free(iter2);
```

```c
    chat_dake_destroy_handshake_data(dataToSend);
    free(dataToSend);
error_with_dske_info:
    free(dske_info);
error:
    return 1;
}

int chat_dske_handle_handshake_message(ChatContextPtr ctx, ChatMessage *msg, OtrlListPtr fnprnt_list , ChatMessage **msgToSend, int *free_msg)
{
    ChatDSKEInfoPtr dske_info;
    ChatMessagePayloadDAKEHandshake *handshake_msg = msg->payload;
    ChatParticipantPtr sender;
    DAKE *sender_dake;
    DAKE_confirm_message_data *dataToSend;
    unsigned int pos;
    unsigned char *fingerprint;
    OtrlChatFingerprintPtr fnprnt;
    int err;

    fprintf(stderr,"chat_dske_handle_handshake_message: start\n");

    dske_info = chat_context_get_dske_info(ctx);

    sender = chat_participant_find(chat_context_get_participants_list(ctx), msg->senderName, &pos);
    if(!sender) { goto error; }

    if(NULL == dske_info || CHAT_DSKESTATE_NONE == dske_info->state){ goto error; }

    sender_dake = chat_participant_get_dake(sender);

    /* If we were not expecting a handshake from this user return err */
    if(!sender_dake || DAKE_STATE_WAITING_HANDSHAKE != sender_dake->state) { goto error; }

    /* Load the keys they sent us and determine what data we should send them */
    err = chat_dake_load_their_part(sender_dake, handshake_msg->handshake_data, &dataToSend, &fingerprint);
    if(err) { goto error; }

    /* Check if the fingerprint calculated during the dake exists in the list of known fingerprints
     * if not, add a new fingerprint in the list */
    fnprnt = chat_fingerprint_find(fnprnt_list, chat_context_get_accountname(ctx), chat_context_get_protocol(ctx) , chat_participant_get_username(sender), fingerprint);

    if (NULL == fnprnt) {
        fnprnt = chat_fingerprint_new(chat_context_get_accountname(ctx), chat_context_get_protocol(ctx), chat_participant_get_username(sender), fingerprint, 0);
        if(!fnprnt) { goto error_with_fingerprint; }

        err = chat_fingerprint_add(fnprnt_list, fnprnt);
        if(err) { goto error_with_fnprnt; }
    }

    /* Set a reference to the user's fingerprint */
    chat_participant_set_fingerprint(sender, fnprnt);

    /* Create the message we should send */
    *msgToSend = chat_message_dake_confirm_new(ctx, pos, dataToSend);
    if(!*msgToSend) { goto error_with_fingerprint; }

    free(fingerprint);

    fprintf(stderr,"chat_dske_handle_handshake_message: end\n");
```

```
    return 0;

error_with_fnprnt:
    chat_fingerprint_free(fnprnt);
error_with_fingerprint:
    free(fingerprint);
    chat_dake_destroy_confirm_data(dataToSend);
    free(dataToSend);
error:
    return 1;
}

int chat_dske_handle_confirm_message(ChatContextPtr ctx, ChatMessage *msg,
    ChatMessage **msgToSend, int *free_msg)
{
    ChatDSKEInfoPtr dske_info;
    ChatMessagePayloadDAKEConfirm *confirm_msg = msg->payload;
    DAKE_key_message_data *dataToSend;
    unsigned char *key_bytes = NULL;
    size_t key_len;
    ChatParticipantPtr sender;
    unsigned int their_pos;
    unsigned int our_pos;
    int err;

    fprintf(stderr,"libotr-mpOTR: chat_dske_handle_confirm_message:
    start\n");

    dske_info = chat_context_get_dske_info(ctx);

    sender =
    chat_participant_find(chat_context_get_participants_list(ctx),
    msg->senderName, &their_pos);
    if(!sender) { goto error; }

    err =
    chat_participant_get_position(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &our_pos);
    if(err) { goto error; }

    if(confirm_msg->recipient != our_pos) { goto error; }

    /* Check if we shouldn't have received this message */
    if(CHAT_DSKESTATE_AWAITING_KEYS != dske_info->state) { goto error; }

    if(DAKE_STATE_WAITING_CONFIRM !=
    chat_participant_get_dake(sender)->state){ goto error; }

    /* Verify that we have computed the same shared secret */
    err = chat_dake_verify_confirm(chat_participant_get_dake(sender),
    confirm_msg->data->mac);
    if(err) { goto error; }

    /* Get the serialized pubkey */
    err = chat_sign_serialize_pubkey(chat_context_get_signing_key(ctx),
    &key_bytes, &key_len);
    if(err) { goto error; }

    /* Encrypt and authenticate our pubkey to send the to the other party
    */
    err = chat_dake_send_key(chat_participant_get_dake(sender), key_bytes,
    key_len, &dataToSend);
    if(err) { goto error_with_key_bytes; }

    /* Create the message to send */
    *msgToSend = chat_message_dake_key_new(ctx, their_pos, dataToSend);
    if(!*msgToSend) { goto error_with_key_bytes; }

    free(key_bytes);

    fprintf(stderr,"chat_dske_handle_confirm_message: end\n");

```

```
    return 0;

error_with_key_bytes:
    free(key_bytes);
error:
    return 1;
}

int chat_dske_handle_key_message(ChatContextPtr ctx, ChatMessage *msg,
                                 ChatMessage **msgToSend, int *free_msg)
{
    ChatDSKEInfoPtr dske_info;
    ChatMessagePayloadDAKEKey *key_msg = msg->payload;
    ChatParticipantPtr sender;
    SignKey *sign_key;
    unsigned char *plain_key;
    size_t keylen;
    unsigned int our_pos, their_pos;
    int err;

    fprintf(stderr,"chat_dske_handle_key_message: start\n");

    dske_info = chat_context_get_dske_info(ctx);

    sender =
    chat_participant_find(chat_context_get_participants_list(ctx),
    msg->senderName, &their_pos);
    if(!sender) { goto error; }

    err =
    chat_participant_get_position(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &our_pos);
    if(err) { goto error; }

    if(key_msg->recipient !=  our_pos) { goto error; }

    if(CHAT_DSKESTATE_AWAITING_KEYS != dske_info->state) { goto error; }

    if(DAKE_STATE_WAITING_KEY != chat_participant_get_dake(sender)->state)
    { goto error; }

    err = chat_dake_receive_key(chat_participant_get_dake(sender),
    key_msg->data, &plain_key, &keylen);
    if(err) { goto error; }

    sign_key = chat_sign_parse_pubkey(plain_key, keylen);
    free(plain_key);
    if(!sign_key) { goto error; }

    chat_participant_set_sign_key(sender, sign_key);

    chat_participant_get_dake(sender)->state = DAKE_STATE_DONE;
    dske_info->remaining -= 1;
    if(dske_info->remaining == 0) {
        dske_info->state = CHAT_DSKESTATE_FINISHED;
    }

    fprintf(stderr,"chat_dske_handle_key_message: end\n");
    return 0;

error:
    return 1;

}

int chat_dske_handle_message(ChatContextPtr ctx, ChatMessage *msg,
    OtrlListPtr fnprnt_list,
                             ChatMessage **msgToSend) {
    ChatMessageType msgType = msg->msgType;
    int free_msg;

    switch(msgType) {
```

```
        case CHAT_MSGTYPE_DAKE_HANDSHAKE:
            return chat_dske_handle_handshake_message(ctx, msg,
    fnprnt_list, msgToSend, &free_msg);
        case CHAT_MSGTYPE_DAKE_CONFIRM:
            return chat_dske_handle_confirm_message(ctx, msg, msgToSend,
    &free_msg);
        case CHAT_MSGTYPE_DAKE_KEY:
            return chat_dske_handle_key_message(ctx, msg, msgToSend,
    &free_msg);
        default:
            return 1;
    }
}

int chat_dske_is_my_message(const ChatMessage *msg)
{
    ChatMessageType msg_type = msg->msgType;

    switch(msg_type) {
    case CHAT_MSGTYPE_DAKE_HANDSHAKE:
    case CHAT_MSGTYPE_DAKE_CONFIRM:
    case CHAT_MSGTYPE_DAKE_KEY:
            return 1;
        default:
            return 0;
    }
}
```

Listing Γ′.2: chat_dske.c

Appendix Δ′

GKA source code

```
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros
    <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos
    <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
    02110-1301 USA
 */

#ifndef CHAT_GKA_H_
#define CHAT_GKA_H_

#include "chat_types.h"

typedef enum {
        CHAT_GKASTATE_NONE,
        CHAT_GKASTATE_AWAITING_UPFLOW,
        CHAT_GKASTATE_AWAITING_DOWNFLOW,
        CHAT_GKASTATE_FINISHED
} ChatGKAState;

struct OtrlListOpsStruct interKeyOps;

unsigned int chat_gka_info_get_position(ChatGKAInfoPtr gka_info);
ChatGKAState chat_gka_info_get_state(ChatGKAInfoPtr gka_info);

void chat_gka_info_free(ChatGKAInfoPtr gka_info);

int chat_gka_init(ChatContextPtr ctx, ChatMessage **msgToSend);

int chat_gka_is_my_message(const ChatMessage *msg);

int chat_gka_handle_message(ChatContextPtr ctx, ChatMessage *msg,
        ChatMessage **msgToSend);

#endif /* CHAT_GKA_H_ */
```

LISTING Δ′.1: chat_gka.h

```
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros
    <dim.kolotouros@gmail.com>,
```

```c
 *                           Konstantinos Andrikopoulos <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 */

#include "chat_gka.h"

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "chat_context.h"
#include "chat_enc.h"
#include "chat_message.h"
#include "chat_participant.h"
#include "chat_types.h"
#include "dh.h"
#include "list.h"

struct ChatGKAInfo {
        ChatGKAState state;  /* the gka state */
        unsigned int position;      /* Our position in the participants order starting from the gka initiator */
        DH_keypair *keypair;        /* The keypair used for the gka */
        unsigned char participants_hash[CHAT_PARTICIPANTS_HASH_LENGTH];
};

ChatGKAInfoPtr chat_gka_info_new()
{
    ChatGKAInfoPtr gka_info;

    gka_info = malloc(sizeof *gka_info);
    if(!gka_info) {
        return NULL;
    }

    gka_info->keypair = NULL;
    gka_info->state = CHAT_GKASTATE_NONE;

    return gka_info;
}

unsigned int chat_gka_info_get_position(ChatGKAInfoPtr gka_info)
{
    return gka_info->position;
}

ChatGKAState chat_gka_info_get_state(ChatGKAInfoPtr gka_info)
{
    return gka_info->state;
}

void chat_gka_info_free(ChatGKAInfoPtr gka_info)
{
    if(!gka_info){
        return;
    }

```

```
    if(gka_info->keypair)
        otrl_dh_keypair_free(gka_info->keypair);

    free(gka_info->keypair);
    free(gka_info);
}

int chat_gka_keys_compareOp(OtrlListPayloadPtr a, OtrlListPayloadPtr b)
{
    return gcry_mpi_cmp(a, b);
}

void chat_gka_key_freeOp(OtrlListPayloadPtr a)
{
    gcry_mpi_t *w = a;

    gcry_mpi_release(*w);

    free(a);
}

void chat_gka_key_printOp(OtrlListNodePtr node)
{
    gcry_mpi_t *w = otrl_list_node_get_payload(node);
    unsigned char *buf;
    size_t s;

    gcry_mpi_print(GCRYMPI_FMT_HEX,NULL,0,&s,*w);
    buf = malloc((s+1) * sizeof *buf);
    gcry_mpi_print(GCRYMPI_FMT_HEX,buf,s,NULL,*w);
    buf[s]='\0';

    fprintf(stderr, "Intermediate key:\n");
    fprintf(stderr, "|- value\t: %s\n", buf);
    free(buf);
}

void chat_gka_mpi_print(gcry_mpi_t w)
{
    unsigned char *buf;
    size_t s;

    gcry_mpi_print(GCRYMPI_FMT_HEX,NULL,0,&s,w);
    buf = malloc((s+1) * sizeof *buf);
    gcry_mpi_print(GCRYMPI_FMT_HEX,buf,s,NULL,w);
    buf[s]='\0';

    fprintf(stderr, "%s\n", buf);
    free(buf);
}

struct OtrlListOpsStruct interKeyOps = {
        chat_gka_keys_compareOp,
        chat_gka_key_printOp,
        chat_gka_key_freeOp
};

OtrlListPtr chat_gka_initial_intermediate_key_list()
{
    OtrlListPtr key_list;
    gcry_mpi_t *generator;
    OtrlListNodePtr node;

    generator = malloc(sizeof *generator);
    if(!generator) {
        return NULL;
    }

    /* Get the generator of the group */
    *generator = gcry_mpi_copy(otrl_dh_get_generator());

    /* Initialize a new list and check if it was actually initialized */
```

```
    key_list = otrl_list_new(&interKeyOps, sizeof(gcry_mpi_t));
    if(!key_list) { goto error; }

    /* Append the generator in the list and check if it was inserted correctly */
    node = otrl_list_append(key_list, generator);
    if(!node) { goto error_with_list; }

    return key_list;

error_with_list:
    otrl_list_free(key_list);
error:
    gcry_mpi_release(*generator);
    free(generator);
    return NULL;

}

int chat_gka_append_with_key(OtrlListPtr new_key_list, OtrlListPtr old_key_list, DH_keypair *key)
{
    OtrlListIteratorPtr iter;
    OtrlListNodePtr cur, node;
    gcry_mpi_t *w, *tmp;
    int err;

    /* For every key in the key_list raise it to the key->priv
     * and append it to the new_list */
    iter = otrl_list_iterator_new(old_key_list);
    if(!iter) { goto error; }
    while(otrl_list_iterator_has_next(iter)) {
        cur = otrl_list_iterator_next(iter);
        tmp = otrl_list_node_get_payload(cur);

        err = otrl_dh_is_inrange(*tmp);
        if(err) { goto error_with_iter; }

        /* Allocate a new gcry_mpi_t to be held in the list */
        w = malloc(sizeof *w);
        if(!w) { goto error_with_iter; }
        *w = gcry_mpi_new(256);

        /* raise it to the key->priv (mod the modulo) */
        otrl_dh_powm(*w, *tmp , key->priv);

        /* Append it to the new_list and check if it was added correctly */
        node = otrl_list_append(new_key_list,w);
        if(!node){ goto error_with_w; };
    }

    otrl_list_iterator_free(iter);

    return 0;

error_with_w:
    gcry_mpi_release(*w);
    free(w);
error_with_iter:
    otrl_list_iterator_free(iter);
error:
    return 1;
}

OtrlListPtr chat_gka_intermediate_key_list_to_send(OtrlListPtr key_list, DH_keypair *key)
{
    OtrlListPtr new_list;
    OtrlListNodePtr node;
    gcry_mpi_t *w, *last, *first;
    int err;

```

```
    /* If the intermediate key_list we received is from the first upflow
    message
     * we should check that the sender is using the correct generator */
    node = otrl_list_get_head(key_list);
    if(!node) { goto error; }

    first = otrl_list_node_get_payload(node);
    if(!first) { goto error; }

    if(2 == otrl_list_size(key_list) &&
    gcry_mpi_cmp(otrl_dh_get_generator(),*first)){
        goto error;
     }

    /* Append the last key in the key_list to the new list, as
     * specified by the algorithm */
    node = otrl_list_get_tail(key_list);
    if(!node) { goto error; }

    last = otrl_list_node_get_payload(node);
    if(!last) { goto error; }

    /* Initialize the list to be returned */
    new_list = otrl_list_new(&interKeyOps, sizeof(gcry_mpi_t));
    if(!new_list) { goto error; }

    w = malloc(sizeof *w);
    if(!w) { goto error_with_new_list; }

    *w = gcry_mpi_copy(*last);

    node = otrl_list_append(new_list, w);
    if(!node) { goto error_with_w; }

    /* If there was an error destroy the new_list and return NULL */
    err = chat_gka_append_with_key(new_list, key_list, key);
    if(err) { goto error_with_new_list; }

    otrl_list_dump(new_list);

    return new_list;

error_with_w:
    gcry_mpi_release(*w);
    free(w);
error_with_new_list:
    otrl_list_free(new_list);
error:
    return NULL;
}

OtrlListPtr chat_gka_final_key_list_to_send(OtrlListPtr key_list,
    DH_keypair *key)
{
    OtrlListPtr new_list;
    int err;

    /* Create a new list */
    new_list = otrl_list_new(&interKeyOps, sizeof(gcry_mpi_t));
    if(!new_list) { goto error; }

    /* And add each intermediate key, raising it to our private key */
    err = chat_gka_append_with_key(new_list, key_list, key);
    if(err) { goto error_with_new_list; }

    return new_list;

error_with_new_list:
    otrl_list_free(new_list);
error:
    return NULL;
}
```

```
gcry_error_t chat_gka_get_participants_hash(OtrlListPtr participants, unsigned char* hash)
{
    gcry_md_hd_t md;
    gcry_error_t err;
    OtrlListIteratorPtr iter;
    OtrlListNodePtr cur;
    ChatParticipantPtr participant;
    size_t len;
    unsigned char *hash_result;

    /* Open a new md */
    err = gcry_md_open(&md, GCRY_MD_SHA512, 0);
    if(err) { goto error; }

    /* Iterate over the list and write each username in the message digest */
    iter = otrl_list_iterator_new(participants);
    if(!iter) { goto error; }
    while(otrl_list_iterator_has_next(iter)) {
        cur = otrl_list_iterator_next(iter);
        participant = otrl_list_node_get_payload(cur);
        len = strlen(chat_participant_get_username(participant));
        gcry_md_write(md, chat_participant_get_username(participant), len);
    }

    gcry_md_final(md);
    hash_result = gcry_md_read(md, GCRY_MD_SHA512);

    memcpy(hash, hash_result, CHAT_PARTICIPANTS_HASH_LENGTH);

    gcry_md_close(md);
    otrl_list_iterator_free(iter);

    return gcry_error(GPG_ERR_NO_ERROR);

error:
    return 1;
}

int chat_gka_info_init(ChatGKAInfoPtr gka_info)
{
    gcry_error_t err;

    /* Allocate the DH keypair */
    if(gka_info->keypair)
        free(gka_info->keypair);

    gka_info->keypair = malloc(sizeof *(gka_info->keypair));
    if(!gka_info->keypair) { goto error; }

    /* Initialize it */
    otrl_dh_keypair_init(gka_info->keypair);

    /* Generate a key */
    err = otrl_dh_gen_keypair(DH1536_GROUP_ID, gka_info->keypair);
    if(err) { goto error; }

    return 0;

error:
    return 1;
}

int chat_gka_init(ChatContextPtr ctx, ChatMessage **msgToSend)
{
    ChatGKAInfoPtr gka_info;
    ChatMessage *newmsg = NULL;
    unsigned int me_next[2];
    gcry_error_t g_err;
    int err;
```

```
    OtrlListPtr inter_key_list;
    OtrlListPtr initial_key_list;

    fprintf(stderr, "libotr-mpOTR: chat_gka_init: start\n");

    gka_info = chat_gka_info_new();
    if(!gka_info) { goto error; }

    /* Initialize the gka info */
    g_err = chat_gka_info_init(gka_info);
    if(g_err) { goto error_with_gka_info; }

    /* Get our position in the upflow stream */
    err =
    chat_participant_get_me_next_position(chat_context_get_accountname(ctx),
    chat_context_get_participants_list(ctx), me_next);
    if(err) { goto error_with_gka_info; }

    if(0 != me_next[0]){
        gka_info->state = CHAT_GKASTATE_AWAITING_UPFLOW;
        newmsg = NULL;

    } else {
        /* Get a intermediate key list with only the generator inside */
        initial_key_list = chat_gka_initial_intermediate_key_list();
        if(!initial_key_list) { goto error_with_gka_info; }

        /* Create the intermediate key list to send */
        inter_key_list =
    chat_gka_intermediate_key_list_to_send(initial_key_list,
    gka_info->keypair);
        if(!inter_key_list) { goto error_with_initial_key_list; }

        /* Generate the message that will be sent */
        newmsg = chat_message_gka_upflow_new(ctx, inter_key_list,
    me_next[1]);
        if(!newmsg) { goto error_with_inter_list; }

        /* Set the gka_info state to await downflow message */
        gka_info->state = CHAT_GKASTATE_AWAITING_DOWNFLOW;
        gka_info->position = 0;

        otrl_list_free(initial_key_list);
    }

    chat_context_set_gka_info(ctx, gka_info);

    *msgToSend = newmsg;
    fprintf(stderr, "libotr-mpOTR: chat_gka_init: end\n");

    return 0;

error_with_inter_list:
    otrl_list_free(inter_key_list);
error_with_initial_key_list:
    otrl_list_free(initial_key_list);
error_with_gka_info:
    chat_gka_info_free(gka_info);
error:
    return 1;
}


int chat_gka_handle_upflow_message(ChatContextPtr ctx, ChatMessage *msg,
     ChatMessage **msgToSend, int *free_msg)
{
    ChatGKAInfoPtr gka_info;
    ChatEncInfo *enc_info;
    ChatMessage *newmsg = NULL;
    OtrlListPtr inter_key_list;
    OtrlListNodePtr last;
    gcry_mpi_t *last_key;
```

```
    ChatMessagePayloadGKAUpflow *upflowMsg;
    unsigned int me_next[2];
    unsigned int inter_key_list_length, participants_list_length;
    gcry_error_t err;

    fprintf(stderr, "libotr-mpOTR: handle_upflow_message: start\n");

    upflowMsg = msg->payload;

    gka_info = chat_context_get_gka_info(ctx);
    if(!gka_info) { goto error; }

    if(CHAT_GKASTATE_AWAITING_DOWNFLOW == chat_gka_info_get_state(gka_info)) { goto error; }
    if(CHAT_GKASTATE_NONE == chat_gka_info_get_state(gka_info)) { goto error ; }

    /* Do any initializations needed */
    err = chat_gka_info_init(gka_info);
    if(err) { goto error; }

    /* Get our position in the participants list */
    err = chat_participant_get_me_next_position(chat_context_get_accountname(ctx), chat_context_get_participants_list(ctx), me_next);
    if(err){ goto error_with_gka_init; }

    /* Check if the message is intended for the same users */
    if(upflowMsg->recipient != me_next[0]) { goto error_with_gka_init; }

    /* Get the length of the intermediate keys */
    inter_key_list_length = otrl_list_size(upflowMsg->interKeys);

    gka_info->position = inter_key_list_length - 1;//me_next[0];

    participants_list_length = otrl_list_size(chat_context_get_participants_list(ctx));

    if(inter_key_list_length < participants_list_length ) {
        fprintf(stderr, "libotr-mpOTR: handle_upflow_message: in upflow generation\n");

        /* Generate the intermediate key list that we will send */
        inter_key_list = chat_gka_intermediate_key_list_to_send(upflowMsg->interKeys, gka_info->keypair);
        if(!inter_key_list) { goto error_with_gka_init; }

        newmsg = chat_message_gka_upflow_new(ctx, inter_key_list, me_next[1]);
        if(!newmsg) { goto error_with_inter_key_list; }

        /* Set the gka_info state to await downflow message */
        gka_info->state = CHAT_GKASTATE_AWAITING_DOWNFLOW;
    }
    else {
        /* Get last intermediate key */
        last = otrl_list_get_tail(upflowMsg->interKeys);
        last_key = otrl_list_node_get_payload(last);

        /* Initialize enc_info struct */
        enc_info = chat_enc_info_new();
        if(!enc_info) { goto error_with_gka_init; }

        /* Generate the master secret */
        err = chat_enc_create_secret(enc_info, *last_key, gka_info->keypair);
        if(err) { goto error_with_gka_init; }

        chat_context_set_enc_info(ctx, enc_info);

        /* Drop the last element */
```

```
        otrl_list_remove_and_free(upflowMsg->interKeys, last);

        /* Get the final intermediate key list to send in the downflow message */
        inter_key_list = chat_gka_final_key_list_to_send(upflowMsg->interKeys, gka_info->keypair);
        if(!inter_key_list) { goto error_with_gka_init; }

        fprintf(stderr, "libotr-mpOTR: handle_upflow_message: dumping inter_key_list:\n");
        otrl_list_dump(inter_key_list);

        /* Generate the downflow message */
        newmsg = chat_message_gka_downflow_new(ctx, inter_key_list);
        if(!newmsg) { goto error_with_inter_key_list; }

        /* Set the gka_info state to finished */
        gka_info->state = CHAT_GKASTATE_FINISHED;
        chat_context_set_id(ctx, gka_info->position);
    }

    fprintf(stderr, "libotr-mpOTR: handle_upflow_messaget: end\n");

    *msgToSend = newmsg;
    return 0;

error_with_inter_key_list:
    otrl_list_free(inter_key_list);
error_with_gka_init:
    otrl_dh_keypair_free(gka_info->keypair);
error:
    return 1;
}

int chat_gka_handle_downflow_message(ChatContextPtr ctx, ChatMessage *msg, ChatMessage **msgToSend, int *free_msg)
{
    ChatGKAInfoPtr gka_info;
    ChatEncInfo *enc_info;
    ChatMessagePayloadGKADownflow *downflowMsg;
    OtrlListNodePtr cur;
    gcry_mpi_t *w;
    unsigned int i;
    unsigned int key_list_length;
    gcry_error_t err;

    fprintf(stderr, "libotr-mpOTR: handle_downflow_message: start\n");

    gka_info = chat_context_get_gka_info(ctx);
    if(!gka_info) { goto error;}

    downflowMsg = msg->payload;

    /* Get the intermediate key list length */
    key_list_length = otrl_list_size(downflowMsg->interKeys);

    /* The key list is reversed so we need the i-th element from the end
       of the list. This item is at position key_list_length - gka_info->position */
    i = key_list_length - 1 - gka_info->position;

    /* Get the appropriate intermediate key */
    cur = otrl_list_get(downflowMsg->interKeys, i);
    if(!cur) { goto error; }

    w = otrl_list_node_get_payload(cur);

    /* Initialize enc_info struct */
    enc_info = chat_enc_info_new();
    if(!enc_info) { goto error; }

```

```
    /* Calculate the shared secret */
    err = chat_enc_create_secret(enc_info, *w, gka_info->keypair);
    if(err) { goto error_with_enc_info; }

    // Set enc_info struct to context
    chat_context_set_enc_info(ctx, enc_info);

    gka_info->state = CHAT_GKASTATE_FINISHED;
    chat_context_set_id(ctx, gka_info->position);

    *msgToSend = NULL;

    fprintf(stderr, "libotr-mpOTR: handle_downflow_message: end\n");

    return 0;

error_with_enc_info:
    chat_enc_info_free(enc_info);
error:
    return 1;
}

int chat_gka_is_my_message(const ChatMessage *msg)
{
    ChatMessageType msg_type = msg->msgType;

    switch(msg_type) {
        case CHAT_MSGTYPE_GKA_UPFLOW:
        case CHAT_MSGTYPE_GKA_DOWNFLOW:
            return 1;
        default:
            return 0;
    }
}

int chat_gka_handle_message(ChatContextPtr ctx, ChatMessage *msg,
                            ChatMessage **msgToSend) {
    ChatMessageType msgType = msg->msgType;
    int free_msg;

    switch(msgType) {
        case CHAT_MSGTYPE_GKA_UPFLOW:
            return chat_gka_handle_upflow_message(ctx, msg, msgToSend,
    &free_msg);
        case CHAT_MSGTYPE_GKA_DOWNFLOW:
            return chat_gka_handle_downflow_message(ctx, msg, msgToSend,
    &free_msg);
        default:
            return 1;
    }
}
```

Listing Δ′.2: chat_gka.c

Appendix E′

Attest source code

```c
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 */

#ifndef CHAT_ATTEST_H_
#define CHAT_ATTEST_H_

#include "chat_types.h"

typedef enum {
    CHAT_ATTESTSTATE_NONE,
    CHAT_ATTESTSTATE_AWAITING,
    CHAT_ATTESTSTATE_FINISHED
} ChatAttestState;

ChatAttestState chat_attest_info_get_state(ChatAttestInfoPtr attest_info);

void chat_attest_info_free(ChatAttestInfoPtr attest_info);

int chat_attest_init(ChatContextPtr ctx, ChatMessage **msgToSend);

int chat_attest_handle_message(ChatContextPtr ctx, const ChatMessage *msg, ChatMessage **msgToSend);

int chat_attest_is_my_message(ChatMessage *msg);

#endif /* CHAT_ATTEST_H_ */
```

Listing E′.1: chat_attest.h

```c
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
```

```c
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
    02110-1301 USA
 */

#include "chat_attest.h"

#include <gcrypt.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "chat_context.h"
#include "chat_message.h"
#include "chat_participant.h"
#include "chat_protocol.h"
#include "chat_sign.h"
#include "chat_types.h"
#include "context.h"
#include "list.h"

struct ChatAttestInfo {
    size_t size;
    size_t checked_count;
    unsigned short int *checked;
    ChatAttestState state;
};

ChatAttestInfoPtr chat_attest_info_new(size_t size)
{
    ChatAttestInfoPtr attest_info;

    attest_info = malloc(sizeof *attest_info);
    if(!attest_info) { goto error; }

    attest_info->size = size;
    attest_info->checked_count = 0;

    attest_info->checked = calloc(attest_info->size, sizeof
    *(attest_info->checked));
    if(!attest_info->checked) { goto error_with_attest_info; }

    attest_info->state = CHAT_ATTESTSTATE_AWAITING;

    return attest_info;

error_with_attest_info:
    free(attest_info);
error:
    return NULL;
}

ChatAttestState chat_attest_info_get_state(ChatAttestInfoPtr attest_info)
{
    return attest_info->state;
}

void chat_attest_info_free(ChatAttestInfoPtr attest_info)
{
    if(attest_info) {
        free(attest_info->checked);
    }
```

```c
    free(attest_info);
}

int chat_attest_assoctable_hash(OtrlListPtr participants_list, unsigned char **hash)
{
    OtrlListIteratorPtr iter;
    OtrlListNodePtr cur;
    ChatParticipantPtr participant;
    unsigned char *buf = NULL, *key = NULL;
    gcry_md_hd_t md;
    gcry_error_t g_err;
    int err;
    size_t len;

    g_err = gcry_md_open(&md, GCRY_MD_SHA512, 0);
    if(g_err) { goto error; }

    iter = otrl_list_iterator_new(participants_list);
    if(!iter) { goto error_with_md; }

    while(otrl_list_iterator_has_next(iter)) {
        cur = otrl_list_iterator_next(iter);
        participant = otrl_list_node_get_payload(cur);

        if(NULL == chat_participant_get_sign_key(participant)) { goto error_with_iter; }

        err = chat_sign_serialize_pubkey(chat_participant_get_sign_key(participant), &key, &len);
        if(err) { goto error_with_iter; }

        gcry_md_write(md, key, len);
        free(key);
    }

    buf = malloc(CHAT_ATTEST_ASSOCTABLE_HASH_LENGTH * sizeof *buf);
    if(!buf) { goto error_with_iter; }

    memcpy(buf, gcry_md_read(md, GCRY_MD_SHA512), CHAT_ATTEST_ASSOCTABLE_HASH_LENGTH);

    otrl_list_iterator_free(iter);
    gcry_md_close(md);

    *hash = buf;
    return 0;

error_with_iter:
    otrl_list_iterator_free(iter);
error_with_md:
    gcry_md_close(md);
error:
    return 1;
}

int chat_attest_verify_sid(ChatContextPtr ctx, unsigned char *sid)
{
    int res, eq;

    eq = memcmp(chat_context_get_sid(ctx), sid, CHAT_OFFER_SID_LENGTH);
    res = (eq==0) ? 1 : 0;

    return res;
}

int chat_attest_verify_assoctable_hash(OtrlListPtr participants_list, unsigned char *hash, int *result)
{
    int err, res, eq;
    unsigned char *ourhash;
```

```

    err = chat_attest_assoctable_hash(participants_list, &ourhash);
    if(err) { goto error; }

    eq = memcmp(ourhash, hash, CHAT_ATTEST_ASSOCTABLE_HASH_LENGTH);

    res = (eq==0) ? 1 : 0;

    free(ourhash);

    *result = res;
    return 0;

error:
    return 1;
}

int chat_attest_is_ready(ChatAttestInfoPtr attest_info)
{
    return (attest_info->checked_count == attest_info->size) ? 1 : 0;
}

int chat_attest_verify(ChatContextPtr ctx, unsigned char *sid, unsigned
    char *assoctable_hash, unsigned int part_pos, int *result)
{
    ChatAttestInfoPtr attest_info;
    int err, res;

    attest_info = chat_context_get_attest_info(ctx);
    if(!attest_info) { goto error; }

    if(part_pos >= attest_info->size) { goto error; }

    if(attest_info->checked[part_pos]) {
        attest_info->checked[part_pos] = 0;
        attest_info->checked_count--;
    }

    res = chat_attest_verify_sid(ctx, sid);
    if(res) {
        err = chat_attest_verify_assoctable_hash(
    chat_context_get_participants_list(ctx), assoctable_hash, &res);
        if(err) { goto error; }
    }

    if(res) {
        attest_info->checked[part_pos] = 1;
        attest_info->checked_count++;
    }

    *result = res;
    return 0;

error:
    return 1;

}

int chat_attest_info_init(ChatContextPtr ctx)
{
    size_t size;
    ChatAttestInfoPtr attest_info;

    size = otrl_list_size(chat_context_get_participants_list(ctx));

    attest_info = chat_attest_info_new(size);
    if(!attest_info) { goto error; }

    chat_attest_info_free(chat_context_get_attest_info(ctx));

    chat_context_set_attest_info(ctx, attest_info);

```

```
    return 0;

error:
    return 1;
}

int chat_attest_create_our_message(ChatContextPtr ctx, unsigned int
    our_pos , ChatMessage **msgToSend)
{
    int err;
    unsigned char *assoctable_hash;
    ChatMessage *msg;

    err =
    chat_attest_assoctable_hash(chat_context_get_participants_list(ctx),
    &assoctable_hash);
    if(err) { goto error; }

    msg = chat_message_attest_new(ctx, chat_context_get_sid(ctx),
    assoctable_hash);
    if(!msg) { goto error_with_assoctable_hash; }

    free(assoctable_hash);

    *msgToSend = msg;
    return 0;

error_with_assoctable_hash:
    free(assoctable_hash);
error:
    return 1;
}

int chat_attest_init(ChatContextPtr ctx, ChatMessage **msgToSend)
{
    ChatAttestInfoPtr attest_info;
    int err;
    unsigned int our_pos;
    ChatMessage *ourMsg = NULL;

    attest_info = chat_context_get_attest_info(ctx);
    if(NULL == attest_info) {
        err = chat_attest_info_init(ctx);
        if(err) { goto error; }
        attest_info = chat_context_get_attest_info(ctx);
    }

    err =
    chat_participant_get_position(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &our_pos);
    if(err) { goto error; }

    if(!attest_info->checked[our_pos]) {
        err = chat_attest_create_our_message(ctx, our_pos, &ourMsg);
        if(err) { goto error; }
        attest_info->checked[our_pos] = 1;
        attest_info->checked_count++;
    }

    attest_info->state = CHAT_ATTESTSTATE_AWAITING;

    *msgToSend = ourMsg;
    return 0;

error:
    return 1;

}

int chat_attest_handle_message(ChatContextPtr ctx, const ChatMessage *msg,
    ChatMessage **msgToSend)
{
```

```c
    ChatAttestInfoPtr attest_info;
    OtrlListPtr participants_list;
    char *accountname;
    unsigned int our_pos, their_pos;
    int res, err;
    ChatMessagePayloadAttest *payload;
    ChatMessage *ourMsg = NULL;

    fprintf(stderr, "libotr-mpOTR: chat_attest_handle_message: start\n");

    attest_info = chat_context_get_attest_info(ctx);

    if(!attest_info) {
        err = chat_attest_info_init(ctx);
        if(err) { goto error; }
        attest_info = chat_context_get_attest_info(ctx);
    }

    participants_list = chat_context_get_participants_list(ctx);
    accountname = chat_context_get_accountname(ctx);

    if(msg->msgType != CHAT_MSGTYPE_ATTEST) { goto error; }
    if(attest_info->state != CHAT_ATTESTSTATE_AWAITING) { goto error; }

    payload = msg->payload;

    err = chat_participant_get_position(participants_list, msg->senderName, &their_pos);
    if(err) { goto error; }

    err = chat_attest_verify(ctx, payload->sid, payload->assoctable_hash, their_pos, &res);
    if(err) { goto error; }

    if(res == 0) {
        fprintf(stderr, "libotr-mpOTR: chat_attest_handle_message: attest verification failed for participant #: %u\n", their_pos);
        chat_protocol_reset(ctx);
        goto error;
    } else {

        // Create our attest message if we haven't already sent one
        err = chat_participant_get_position(participants_list, accountname, &our_pos);
        if(err) { goto error; }

        if(!attest_info->checked[our_pos]) {
            err = chat_attest_create_our_message(ctx, our_pos, &ourMsg);
            if(err) { goto error; }

            attest_info->checked[our_pos] = 1;
            attest_info->checked_count++;
        }

        if(chat_attest_is_ready(attest_info)) {
            attest_info->state = CHAT_ATTESTSTATE_FINISHED;
            chat_context_set_msg_state(ctx, OTRL_MSGSTATE_ENCRYPTED);
        }
    }

    fprintf(stderr, "libotr-mpOTR: chat_attest_handle_message: end\n");

    *msgToSend = ourMsg;
    return 0;

error:
    return 1;
}


int chat_attest_is_my_message(ChatMessage *msg)
{
```

```
    ChatMessageType msg_type = msg->msgType;

    switch(msg_type) {
        case CHAT_MSGTYPE_ATTEST:
            return 1;
        default:
            return 0;
    }
}
```

LISTING E′.2: chat_attest.c

Appendix ΣΤ′

Communication source code

```
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros
    <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos
    <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
    02110-1301 USA
 */

#ifndef CHAT_COMMUNICATION_H_
#define CHAT_COMMUNICATION_H_

int chat_communication_broadcast(ChatContextPtr ctx, const char *message,
    ChatMessage **msgToSend);
int chat_communication_is_my_message(ChatMessage *msg);
int chat_communication_handle_msg(ChatContextPtr ctx, ChatMessage *msg,
    ChatMessage **msgToSend, char **plaintext);


#endif /* CHAT_COMMUNICATION_H_ */
```

Listing ΣΤ′.1: chat_communication.h

```
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros
    <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos
    <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
    02110-1301 USA
```

```c
 */

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "context.h"
#include "chat_context.h"
#include "chat_enc.h"
#include "chat_message.h"
#include "chat_participant.h"
#include "chat_types.h"
#include "list.h"

int chat_communication_handle_data_message(ChatContextPtr ctx, ChatMessage *msg,
                                            ChatMessage **msgToSend, char** plaintext)
{
    ChatMessagePayloadData *payload = msg->payload;
    ChatParticipantPtr sender;
    OtrlListNodePtr node;
    unsigned int sender_pos;
    char *plain = NULL;
    char *plain_cpy = NULL;

    fprintf(stderr, "libotr-mpOTR: chat_communication_handle_data_message: start\n");

    switch(chat_context_get_msg_state(ctx)) {

        case OTRL_MSGSTATE_PLAINTEXT:
        case OTRL_MSGSTATE_FINISHED:
            goto error;
            break;

        case OTRL_MSGSTATE_ENCRYPTED:

            plain = chat_enc_decrypt(ctx, payload->ciphertext, payload->datalen, payload->ctr, msg->senderName);
            if (!plain) { goto error; }

            sender = chat_participant_find(chat_context_get_participants_list(ctx), msg->senderName, &sender_pos);
            if(!sender) {goto error_with_plain; }

            plain_cpy = strdup(plain);
            if(!plain_cpy) { goto error_with_plain; }

            node = otrl_list_insert(chat_participant_get_messages(sender), plain_cpy);
            if(!node) { goto error_with_copy; }

            otrl_list_dump(chat_participant_get_messages(sender));
            break;
    }

    *plaintext = plain;

    fprintf(stderr, "libotr-mpOTR: chat_communication_handle_data_message: end\n");

    return 0;

error_with_copy:
    free(plain_cpy);
error_with_plain:
    free(plain);
error:
    return 1;
```

```
}

int chat_communication_broadcast(ChatContextPtr ctx, const char *message,
                                 ChatMessage **msgToSend)
{
    unsigned char *ciphertext;
    OtrlListNodePtr node;
    size_t datalen;
    ChatMessage *msg = NULL;
    ChatParticipantPtr me;
    unsigned int pos;
    char *msg_cpy = NULL;

    fprintf(stderr, "libotr-mpOTR: chat_communication_broadcast: start\n");

    /* Find the user in the participants list */
    me = chat_participant_find(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &pos);
    if(!me) { goto error; }

    /* Copy the message to send */
    msg_cpy = strdup(message);
    if(!msg_cpy) { goto error; }

    ciphertext = chat_enc_encrypt(ctx, message);
    if(!ciphertext) { goto error_with_msg_cpy; }

    datalen = strlen(message);

    msg = chat_message_data_new(ctx, chat_context_get_enc_info(ctx)->ctr,
    datalen, ciphertext);
    if(!msg) { goto error_with_ciphertext; }

    node = otrl_list_insert(chat_participant_get_messages(me), msg_cpy);
    if(!node) { goto error_with_msg; }

    otrl_list_dump(chat_participant_get_messages(me));

    fprintf(stderr, "libotr-mpOTR: chat_communication_broadcast: end\n");

    *msgToSend = msg;
    return 0;

error_with_msg:
    chat_message_free(msg);
error_with_ciphertext:
    free(ciphertext);
error_with_msg_cpy:
    free(msg_cpy);
error:
    return 1;
}

int chat_communication_is_my_message(ChatMessage *msg)
{
    ChatMessageType msg_type = msg->msgType;

    switch(msg_type) {
        case CHAT_MSGTYPE_DATA:
            return 1;
        default:
            return 0;
    }

}

int chat_communication_handle_msg(ChatContextPtr ctx, ChatMessage *msg,
                                  ChatMessage **msgToSend, char
    **plaintext)
{
    ChatMessageType msg_type = msg->msgType;
```

```
    int err;
    char *plain;

    switch(msg_type) {
        case CHAT_MSGTYPE_DATA:
            err = chat_communication_handle_data_message(ctx, msg,
    msgToSend, &plain);
            if(err) { goto error; }
            break;
        default:
            goto error;
    }

    *plaintext = plain;
    return 0;

error:
    return 1;
}
```

Listing ΣT′.2: chat_communication.c

Appendix Z′

Shutdown source code

```c
#ifndef CHAT_SHUTDOWN_H
#define CHAT_SHUTDOWN_H

#include "chat_types.h"

typedef enum {
    CHAT_SHUTDOWNSTATE_NONE,
    CHAT_SHUTDOWNSTATE_AWAITING_SHUTDOWNS,
    CHAT_SHUTDOWNSTATE_AWAITING_DIGESTS,
    CHAT_SHUTDOWNSTATE_AWAITING_ENDS,
    CHAT_SHUTDOWNSTATE_FINISHED
} ChatShutdownState;

ChatShutdownState chat_shutdown_info_get_state(ChatShutdownInfoPtr shutdown_info);
void chat_shutdown_info_free(ChatShutdownInfoPtr shutdown_info);

int chat_shutdown_init(ChatContextPtr ctx);

int chat_shutdown_send_shutdown(ChatContextPtr ctx, ChatMessage **msgToSend);

int chat_shutdown_send_digest(ChatContextPtr ctx, ChatMessage **msgToSend);

int chat_shutdown_send_end(ChatContextPtr ctx, ChatMessage **msgToSend);

int chat_shutdown_release_secrets(ChatContextPtr ctx, ChatMessage **msgToSend);

int chat_shutdown_is_my_message(const ChatMessage *msg);

int chat_shutdown_handle_message(ChatContextPtr ctx, ChatMessage *msg,
                                 ChatMessage **msgToSend);

#endif /* CHAT_SHUTDOWN_H */
```

Listing Z′.1: chat_shutdown.h

```c
/*
 *  Off-the-Record Messaging library
 *  Copyright (C) 2015-2016  Dimitrios Kolotouros <dim.kolotouros@gmail.com>,
 *                           Konstantinos Andrikopoulos <el11151@mail.ntua.gr>
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of version 2.1 of the GNU Lesser General
 *  Public License as published by the Free Software Foundation.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
```

```c
 *  Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 */

#include "chat_shutdown.h"

#include "chat_context.h"
#include "chat_message.h"
#include "chat_participant.h"
#include "chat_sign.h"
#include "chat_types.h"
#include "list.h"

#define CONSENSUS_HASH_LEN 64

struct ChatShutdownInfo{
    int shutdowns_remaining;
    int digests_remaining;
    int ends_remaining;
    unsigned char *has_send_end;
    unsigned char *consensus_hash;
    ChatShutdownState state;
};

int get_consensus_hash(OtrlListPtr participants_list, unsigned char *result)
{
    gcry_md_hd_t md;
    gcry_error_t err;
    OtrlListIteratorPtr iter;
    OtrlListNodePtr cur;
    ChatParticipantPtr participant;
    size_t len;
    unsigned char *hash_result = NULL;

    /* Open a digest */
    err = gcry_md_open(&md, GCRY_MD_SHA512, 0);
    if(err) { goto error; }

    /* Iterate over each participant in partipants_list */
    iter = otrl_list_iterator_new(participants_list);
    if(!iter) { goto error_with_md; }
    while(otrl_list_iterator_has_next(iter)) {
        cur = otrl_list_iterator_next(iter);
        participant = otrl_list_node_get_payload(cur);
        len = MESSAGES_HASH_LEN;
        /* Write the participant's messages_hash to the digest */
        gcry_md_write(md, chat_participant_get_messages_hash(participant), len);
    }

    /* Finalize the digest */
    gcry_md_final(md);
    /* And read the calculated hash */
    hash_result = gcry_md_read(md, GCRY_MD_SHA512);
    if(!hash_result) { goto error_with_iter; }

    /* Copy the result in the output buffer */
    memcpy(result, hash_result, gcry_md_get_algo_dlen(GCRY_MD_SHA512));

    otrl_list_iterator_free(iter);
    gcry_md_close(md);

    return 0;

error_with_iter:
    otrl_list_iterator_free(iter);
error_with_md:
    gcry_md_close(md);
error:
    return 1;
}
```

```
ChatShutdownState chat_shutdown_info_get_state(ChatShutdownInfoPtr
    shutdown_info)
{
    return shutdown_info->state;
}

void chat_shutdown_info_free(ChatShutdownInfoPtr shutdown_info)
{
    if(shutdown_info) {
        free(shutdown_info->has_send_end);
        free(shutdown_info->consensus_hash);
    }
    free(shutdown_info);
}

int chat_shutdown_init(ChatContextPtr ctx)
{
    ChatShutdownInfoPtr shutdown_info;

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_init: start\n");

    shutdown_info = malloc(sizeof *shutdown_info);
    if(!shutdown_info) { goto error; }

    /* Initiliaze the state of each participant */
    shutdown_info->has_send_end =
    calloc(otrl_list_size(chat_context_get_participants_list(ctx)), sizeof
    *shutdown_info->has_send_end);
    if(!shutdown_info->has_send_end){ goto error_with_info; }

    /* We expect a shutdown/digest/end message from all the participants */
    shutdown_info->shutdowns_remaining =
    otrl_list_size(chat_context_get_participants_list(ctx));
    shutdown_info->digests_remaining = shutdown_info->shutdowns_remaining;
    shutdown_info->ends_remaining = shutdown_info->shutdowns_remaining;

    /* The initial state of the protocol is to wait for shutdown messages
    */
    shutdown_info->state = CHAT_SHUTDOWNSTATE_AWAITING_SHUTDOWNS;

    chat_context_set_shutdown_info(ctx, shutdown_info);

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_init: end\n");

    return 0;
error_with_info:
    free(shutdown_info);
error:
    return 1;
}
int chat_shutdown_send_shutdown(ChatContextPtr ctx, ChatMessage
    **msgToSend)
{
    ChatShutdownInfoPtr shutdown_info;
    ChatParticipantPtr me;
    unsigned int my_pos;
    int err;

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_send_shutdown: start\n");

    shutdown_info = chat_context_get_shutdown_info(ctx);
    if(!shutdown_info) { goto error; }

    /* Find us in the participants list */
    me = chat_participant_find(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &my_pos);
    if(!me) { goto error; }

    /* If we have already sent a shutdown return error */
    if(1 <= shutdown_info->has_send_end[my_pos]) { goto error; }

```

```
    /* Calculate our messages hash, and store it for later */
    err = chat_participant_calculate_messages_hash(me,
    chat_participant_get_messages_hash(me));
    if(err) { goto error; }

    /* Create a shutdown message */
    *msgToSend = chat_message_shutdown_shutdown_new(ctx,
    chat_participant_get_messages_hash(me));
    if(!*msgToSend) { goto error; }

    /* We have sent a shutdown message so update our state */
    shutdown_info->has_send_end[my_pos] = 1;

    /* We wait for one less shutdown message since we just sent one */
    shutdown_info->shutdowns_remaining -= 1;

    /* If everybode has sent us a shutdown message then we must proceed to
    the next phase */
    if(0 == shutdown_info->shutdowns_remaining) {
        /* Allocate memory for the consensus hash */
        shutdown_info->consensus_hash = malloc(CONSENSUS_HASH_LEN * sizeof
    *shutdown_info->consensus_hash);
        if(!shutdown_info->consensus_hash) { goto error; }

        /* And calculate the hash itself */
        err =  get_consensus_hash(chat_context_get_participants_list(ctx),
    shutdown_info->consensus_hash);
        if(err) { goto error_with_consensus_hash; }

        /* Set the state to wait for digest messages */
        shutdown_info->state = CHAT_SHUTDOWNSTATE_AWAITING_DIGESTS;
    }

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_send_shutdown: end\n");
    return 0;

error_with_consensus_hash:
    free(shutdown_info->consensus_hash);
error:
    return 1;
}

int chat_shutdown_handle_shutdown_message(ChatContextPtr ctx, ChatMessage
    *msg,
                                 ChatMessage **msgToSend)
{
    ChatShutdownInfoPtr shutdown_info;
    ChatParticipantPtr sender;
    unsigned int their_pos;
    int err;

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_handle_shutdown:
    start\n");

    shutdown_info = chat_context_get_shutdown_info(ctx);
    if(!shutdown_info) { goto error; }

    /* Get the sender from the participants list. If not found return
    error */
    sender =
    chat_participant_find(chat_context_get_participants_list(ctx),
    msg->senderName, &their_pos);
    if(!sender) { goto error; }

    /* Verify that we expected this message */
    if(CHAT_SHUTDOWNSTATE_AWAITING_SHUTDOWNS != shutdown_info->state) {
    goto error; }

    /* If we have already received the shutdown message from this user
    return
       success */
    if(1 <= shutdown_info->has_send_end[their_pos]) { goto error; }
```

```

    /* Remember that this user has sent us a shutdown */
    shutdown_info->has_send_end[their_pos] = 1; // True
    shutdown_info->shutdowns_remaining -= 1;

    /* Hash the participants messages and store them in sender */
    err = chat_participant_calculate_messages_hash(sender,
    chat_participant_get_messages_hash(sender));
    if(err) { goto error; }

    /* Check if we have received shutdown messages from everybody.
       If yes then send a digest */
    if(0 == shutdown_info->shutdowns_remaining) {
        /* Allocate memory for the consensus hash */
        shutdown_info->consensus_hash = malloc(CONSENSUS_HASH_LEN * sizeof
    *shutdown_info->consensus_hash);
        if(!shutdown_info->consensus_hash) { goto error; }

        /* And calculate it */
        err = get_consensus_hash(chat_context_get_participants_list(ctx),
    shutdown_info->consensus_hash);
        if(err) { goto error; }

        /* Set the state so that we wait for digest messages */
        shutdown_info->state = CHAT_SHUTDOWNSTATE_AWAITING_DIGESTS;

    /* If not then we maybe have to send a shutdown message */
    } else {
        err = chat_shutdown_send_shutdown(ctx, msgToSend);
        if(err) { goto error; }
    }

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_handle_shutdown: end\n");
    return 0;

error:
    return 1;
}

int chat_shutdown_send_digest(ChatContextPtr ctx, ChatMessage **msgToSend)
{
    ChatShutdownInfoPtr shutdown_info;
    ChatParticipantPtr me;
    unsigned int my_pos;

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_send_digest: start\n");

    shutdown_info = chat_context_get_shutdown_info(ctx);
    if(!shutdown_info) { goto error; }

    /* Find us in the participants list */
    me = chat_participant_find(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &my_pos);
    if(!me) { goto error; }

    /* If we already sent a digest message return error */
    if(2 <= shutdown_info->has_send_end[my_pos]) { goto error; }

    /* Create a digest message to send */
    *msgToSend = chat_message_shutdown_digest_new(ctx,
    shutdown_info->consensus_hash);
    if(!*msgToSend) { goto error; }

    /* Remember that we sent a digest */
    shutdown_info->has_send_end[my_pos] = 2;

    /* And we now wait for one less shutdown */
    shutdown_info->digests_remaining -= 1;

    /* If there are no more digest messages pending then update the state
    */
    if(0 == shutdown_info->digests_remaining) {
```

```
        /* We now wait for end messages */
        shutdown_info->state = CHAT_SHUTDOWNSTATE_AWAITING_ENDS;
    }

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_send_digest: end\n");
    return 0;

error:
    return 1;
}

int chat_shutdown_handle_digest_message(ChatContextPtr ctx, ChatMessage *msg, ChatMessage **msgToSend)
{
    ChatShutdownInfoPtr shutdown_info;
    ChatMessagePayloadShutdownDigest *digest_msg = msg->payload;
    ChatParticipantPtr sender;
    int consensus;
    unsigned int their_pos;

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_handle_digest_message: start\n");

    shutdown_info = chat_context_get_shutdown_info(ctx);
    if(!shutdown_info) { goto error; }

    /* Get the sender from the participants list. If not found return error */
    sender = chat_participant_find(chat_context_get_participants_list(ctx), msg->senderName, &their_pos);
    if(!sender) { goto error; }

    /* Verify that we expected this message */
    if(CHAT_SHUTDOWNSTATE_AWAITING_DIGESTS != shutdown_info->state) { goto error; }

    /* If we have already received the shutdown message from this user return
       success */
    if(2 <= shutdown_info->has_send_end[their_pos]) { goto error; }

    /* Remember that this user has sent us a digest */
    shutdown_info->has_send_end[their_pos] = 2; // True

    /* We need to wait for one less digest message now */
    shutdown_info->digests_remaining -= 1;

    /* Determine consensus with this user */
    consensus = memcmp(digest_msg->digest, shutdown_info->consensus_hash, CONSENSUS_HASH_LEN) ? 0 : 1;
    chat_participant_set_consensus(sender, consensus);

    fprintf(stderr, "libotr-mpOTR: local digest: ");
    for(int i = 0; i < CONSENSUS_HASH_LEN; i++)
        fprintf(stderr, "%0X", shutdown_info->consensus_hash[i]);
    fprintf(stderr, "\nlibotr-mpOTR: received digest: ");
    for(int i = 0; i < CONSENSUS_HASH_LEN; i++)
        fprintf(stderr, "%0X", digest_msg->digest[i]);
    fprintf(stderr, "\n");


    /* If there are no more pending digest messages update the state */
    if(0 == shutdown_info->digests_remaining) {
        /* We now wait for end messages */
        shutdown_info->state = CHAT_SHUTDOWNSTATE_AWAITING_ENDS;
    }

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_handle_digest_message: end\n");
    return 0;

```

```
error:
    return 1;
}

int chat_shutdown_send_end(ChatContextPtr ctx, ChatMessage **msgToSend)
{
    ChatShutdownInfoPtr shutdown_info;
    ChatParticipantPtr me;
    unsigned int my_pos;

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_send_end: start\n");

    shutdown_info = chat_context_get_shutdown_info(ctx);
    if(!shutdown_info) { goto error; }

    /* Find us in the participant list */
    me = chat_participant_find(chat_context_get_participants_list(ctx),
    chat_context_get_accountname(ctx), &my_pos);
    if(!me) { goto error; }

    /* If we already sent an end message return error */
    if(3 <= shutdown_info->has_send_end[my_pos]) { goto error; }

    /* Create an end message to send */
    *msgToSend = chat_message_shutdown_end_new(ctx);
    if(!*msgToSend) { goto error; }

    /* Remember that we sent an end message */
    shutdown_info->has_send_end[my_pos] = 3;

    /* Decrement the pending end messages */
    shutdown_info->ends_remaining -= 1;

    /* If there are no more pending messages then update the state */
    if(0 == shutdown_info->ends_remaining){
        /* We have finished the shutdown subprotocol */
        shutdown_info->state = CHAT_SHUTDOWNSTATE_FINISHED;
    }

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_send_end: start\n");
    return 0;

error:
    return 1;
}

int chat_shutdown_handle_end_message(ChatContextPtr ctx, ChatMessage *msg,
    ChatMessage **msgToSend)
{
    ChatShutdownInfoPtr shutdown_info;
    ChatParticipantPtr sender;
    unsigned int their_pos;

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_handle_end_message:
    start\n");

    shutdown_info = chat_context_get_shutdown_info(ctx);
    if(!shutdown_info) { goto error; }

    /* Get the sender from the participants list. If not found return
    error */
    sender =
    chat_participant_find(chat_context_get_participants_list(ctx),
    msg->senderName, &their_pos);
    if(!sender) { goto error; }

    /* Verify that we expected this message */
    if(CHAT_SHUTDOWNSTATE_AWAITING_ENDS != shutdown_info->state) { goto
    error; }

    /* If we have already received the shutdown message from this user
    return
```

```
          success */
    if(3 <= shutdown_info->has_send_end[their_pos]) { goto error; }

    /* Hash the participants messages and store them in sender */
    if(chat_participant_calculate_messages_hash(sender, chat_participant_get_messages_hash(sender))) { goto error;  }

    /* Remember that this user has sent us a digest */
    shutdown_info->has_send_end[their_pos] = 3; // True
    shutdown_info->ends_remaining -= 1;

    /* If there are no more pending messages update the state */
    if(0 == shutdown_info->ends_remaining){
        /* We have finished the shutdown protocol */
        shutdown_info->state = CHAT_SHUTDOWNSTATE_FINISHED;
    }

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_handle_end_message: start\n");
    return 0;

error:
    return 1;
}

int chat_shutdown_release_secrets(ChatContextPtr ctx, ChatMessage **msgToSend)
{
    unsigned char *key_bytes = NULL;
    size_t keylen;
    int error = 0;

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_release_secrets: start\n");

    /* Serialize the private part of the signing key */
    error = chat_sign_serialize_privkey(chat_context_get_signing_key(ctx), &key_bytes, &keylen);
    if(error) { goto error; }

    /* Create a key release message */
    *msgToSend = chat_message_shutdown_keyrelease_new(ctx, key_bytes, keylen);
    if(!*msgToSend) { goto error_with_key_bytes; }

    fprintf(stderr, "libotr-mpOTR: chat_shutdown_release_secrets: end\n");

    return 0;

error_with_key_bytes:
    free(key_bytes);
error:
    return 1;
}

int chat_shutdown_is_my_message(const ChatMessage *msg)
{
    ChatMessageType msg_type = msg->msgType;

    switch(msg_type) {
        case CHAT_MSGTYPE_SHUTDOWN_SHUTDOWN:
        case CHAT_MSGTYPE_SHUTDOWN_DIGEST:
        case CHAT_MSGTYPE_SHUTDOWN_END:
            return 1;
        default:
            return 0;
    }
}

int chat_shutdown_handle_message(ChatContextPtr ctx, ChatMessage *msg,
                                 ChatMessage **msgToSend)
{
```

```
    ChatMessageType msgType = msg->msgType;
    switch(msgType) {
        case CHAT_MSGTYPE_SHUTDOWN_SHUTDOWN:
            return chat_shutdown_handle_shutdown_message(ctx, msg,
    msgToSend);
        case CHAT_MSGTYPE_SHUTDOWN_DIGEST:
            return chat_shutdown_handle_digest_message(ctx, msg,
    msgToSend);
        case CHAT_MSGTYPE_SHUTDOWN_END:
            return chat_shutdown_handle_end_message(ctx, msg, msgToSend);
        default:
            return 0;
    }
}
```

Listing Z′.2: chat_shutdown.c

Bibliography

[1] Alex Biryukov et al. "Key Recovery Attacks of Practical Complexity on AES Variants With Up To 10 Rounds". In: (2009).

[2] Alexander Chris and Goldber Ian. "Improved User Authentication in Off-The-Record Messaging". In: (2007).

[3] Nelly Fazio and Antonio Nicolosi. "Cryptographic Accumulators: Definitions, Constructions and Applications". In: (2002).

[4] Niels Ferguson et al. "Improved Cryptanalysis of Rijndael". In: (2000).

[5] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[6] Jelle van den Hooff et al. "Vuvuzela: Scalable Private Messaging Resistant To Traffic Analysis". In: *SOSP* (2015).

[7] Goldberg Ian et al. "Multi-party Off-the-Record Messaging". In: (2009).

[8] T. Kivinen. *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE).* https://www.ietf.org/rfc/rfc3526.txt. 2003.

[9] Ximin Luo and Guy Kloss. "Multi-party Encrypted Messaging Protocol design document".

[10] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vannstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[11] Borisov Nikita, Goldberg Ian, and Brewer Eric. "Off-the-Record Communication, or, Why Not To Use PGP". In: (2004).

[12] Stedman Ryan, Yoshida Kayo, and Goldberg Ian. "A User Study of Off-The-Record Messaging". In: (2008).

[13] Adam Tornhill. *Patterns in C.*

[14] Difie Whitfield and Hellman Martin. "New Directions in Cryptography". In: *IEEE Transaction On Information Theory* 22.6 (1976), pp. 644–654.