FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Improvements to the Off-The-Record Protocol
Student:	Ali Mammadov
Supervisor:	Ing. Josef Kokeš
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2018/19

Instructions

1) Study the current state of the Off-The-Record Protocol.
2) Describe the properties of this protocol, explain its state machine, and discuss the practical feasibility of a computational attack.
3) Propose improvements to the protocol to remove or mitigate the vulnerability.
4) Design a Python library covering the modified protocol.
5) Discuss your results.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 4, 2018

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Bachelor's thesis

Improvements to the Off-The-Record Protocol

Ali Mammadov

Supervisor: Ing. Josef Kokeš

12th May 2018

Acknowledgements

First of all, I would like to mention my supervisor Ing. Josef Kokeš for his patience, guidance and excellent soft and domain skills. Thanks to every person who has ever been my teacher: you and your work helped me become who I am now.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 12th May 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Mammadov, Ali. *Improvements to the Off-The-Record Protocol.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Off-The-Record (OTR) je kryptografický protokol, který poskytuje šifrování pro instant messagingové konverzace. Nejnovější verze OTR používá kombinaci symetrického algoritmu AES se 128bitovým klíčem, Diffie-Hellmanovy výměny klíče s grupou o velikosti 1536 bitů, a hashovací funkce SHA-1. Cílem této práce je změnit protokol tak,aby byl lépe zabezpečený proti neustále rostoucí výpočetní síle potenciálních protivníků. Změny navrhované v této práci umožňují flexibilní volbu sady kryptografických algoritmů a jejich parametrů. Výsledkem je možnost zvolit si požadovanou úroveň bezpečnosti a zjednodušena je i úloha řízení bezpečnosti OTR jako celku.

Klíčová slova instant messaging, OTR, soukromí, popiratelné šifrování

Abstract

Off-The-Record messaging (OTR) is a cryptographic protocol that provides encryption for instant messaging conversations. The latest version of OTR uses a combination of AES symmetric-key algorithm with 128-bit key length, Diffie–Hellman key exchange with 1536-bit group size, and SHA-1 hash function. The goal of this work is to introduce changes to this protocol to make

it better secured against growing computational power of potential adversaries. The changes proposed by this work allow flexible selection of the set of cryptoalgorithms and their parameters. As a result, there is a way to choose desired security level and the overall task of managing security of OTR is simplified.

Keywords instant messaging, OTR, privacy, deniable encryption

Contents

List of Figures

Introduction

With more mass surveillance being performed each year, the need for secure encrypted messaging protocols and platforms is rapidly increasing. While there are quite a few protocols for secure and *authenticated* messaging present today, most of them have an undesired consequence: it is easy to prove the authorship of a given message at *any* time. While it may seem unimportant, the mere possibility of this is sometimes unacceptable to certain groups of users. In fact, it was so important to them that a whole new protocol was created: Off-The-Record (OTR), for which main design goal was providing deniability for the conversation participants while maintaining conversation confidentiality[2], like a private conversation in real life, or off the record in journalism sourcing (hence the name), in contrast with other protocols which allow to prove the fact of communication and the identities of the participants.

OTR has several useful properties:

authentication the parties can be sure they talk to the desired person

encryption the messages are readable only to the intended recipient

perfect forward secrecy a compromise of private keys will not lead to the decryption of previous conversations

malleability *after* a conversation has completed, *anyone* is able to forge a message to appear to have come from one of the participants in the conversation, making it is impossible to prove that a specific message came from a specific person.

However, the protocol's age (first version presented in 2004) starts to show itself: when the protocol was designed, computational cost of breaking Diffie-Hellman key exchange with 1536-bit group size was considered way too high even for the most resourceful adversaries. But recent advances in computational power, efficiency and algorithms are slowly starting to impose a threat to the security of Diffie-Hellman exchange[12], on which OTR confidentiality

depends. To make things even worse, OTR does not have a mechanism to easily change the set of underlying cryptoalgorithms and their parameters.

The objective of this thesis is to add flexibility to the existing OTR protocol, and doing it in a way which will be beneficial both in the short and the long-term.

Problem statement

The current problem of OTR is its use of Diffie-Hellman key exchange with 1536-bit group size and SHA-1 hash function as a fingerprint for the public keys. OTR in its current version does not allow to use Diffie-Hellman with parameters other that those currently mentioned in protocol description nor does it allow to use any other hashing algorithm. A successful attack on SHA-1 has been demonstrated[3] and Diffie-Hellman key exchange with 1536-bit group size is no longer approved for use in new systems by NIST[4]. OTR needs to be modified in a way that will allow both parties to negotiate more secure cryptoalgorithms and their parameters.

Chapter 1

State of the Art

This section describes the basics of OTR protocol and gives an overview of the previous work on its security.

1.1 High-level overview of OTR

The process of establishing encrypted and authenticated OTR conversation consists of two main stages:

Querying one party sends a message indicating that it wants to start using OTR

Authenticated Key Exchange both parties perform *unauthenticated* Diffie-Hellman exchange and once inside the encrypted channel, a mutual authentication is performed

Note: OTR assumes a network model which provides in-order delivery of messages, but allows that some messages may not get delivered at all (for example, if the user disconnects). There may be an active attacker who is allowed to perform a Denial of Service attack, but not to learn the contents of messages.[5]

1.2 Requesting an OTR conversation

There are two ways Alice can inform Bob that she is willing to use the OTR protocol to speak with him: by sending him the OTR Query Message, or by including a special "tag" consisting of whitespace characters in one of her messages to him. Each method also includes a way for Alice to communicate to Bob which versions of the OTR protocol she supports.

The semantics of the OTR Query Message are that Alice is requesting that Bob start an OTR conversation with her (if, of course, he is willing and

able to do so). On the other hand, the semantics of the whitespace tag are that Alice is merely indicating to Bob that she is willing and able to have an OTR conversation with him. If Bob has a policy of "only use OTR when it's explicitly requested", for example, then he would start an OTR conversation upon receiving an OTR Query Message, but would not upon receiving the whitespace tag.[5]

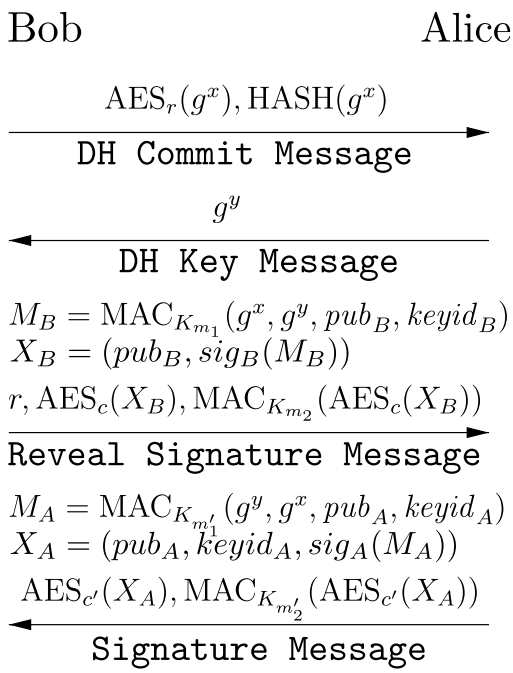

Figure 1.1: OTR Authenticated Key Exchange

1.3 Authenticated Key Exchange

Authentication in OTR is provided by means of long-term public keys and digital signatures. Each participant has a long-term public key which is used for signature generation in the AKE process. Authenticity of public keys should be verified by each participant individually.

During the initial AKE phase, each party authenticates itself and establishes a shared Diffie-Hellman secret. Signatures are used only in the initial AKE stage to allow initial authentication and the deniability in the future. The use of signatures implies that neither party can deny that conversation did in fact occur, only the contents of the conversation are deniable.

Description of the SIGMA protocol[6] used as the AKE is given below. All exponentiations are done modulo a particular 1536-bit prime[7] p_{dh}, and g is a generator of group $\mathbb{Z}^{\times}_{p_{dh}}$. Alice and Bob's long-term authentication public keys are pub_A and pub_B, respectively.

In the figure 1.1[1], Bob is initiating AKE with Alice. First, $r \in \mathbb{Z}_{2^{128}}$ and $x, y \in \mathbb{Z}_{\geq 2^{320}}$ and $g^x, g^y \in [2, p_{dh} - 2]$ have to be satisfied. After DH key message, each party computes s, the shared DH key, in the following way: $s = (g^y)^x = (g^x)^y$. Each party also has to pick *keyid* for its DH key. All signatures have to be verified by the corresponding public keys. MAC and AES keys $c, c', K_{m_1}, K_{m_1'}, K_{m_2}, K_{m_2'}$ are derived from s by hashing it in various ways. *Note*: There are number of checks and constraints omitted in

the diagram for simplicity.

1.4 Exchanging data

Each data message in OTR is encrypted with AES in counter mode, which makes messages malleable to the third party. Malleability is desirable so that parties can later claim that any third party with possession of the proper MAC keys could've produced the messages.

Malleability comes from the fact that AES in counter mode is equivalent to a stream cipher, and the latter does not provide any means for authentication or integrity protection. Authentication is provided by MAC and is done separately, so that the old MAC keys could be discarded along with their corresponding Diffie-Hellman keys. Because the MAC keys are derived by hashing the shared DH key, their publication does not allow a third party to decrypt old messages due to the one-way nature of hash function used, yet it allows anyone to modify the messages and still make them look as valid since now they have all the required information to produce valid MACs for arbitrary data.

1.5 Forward secrecy

Alice and Bob are constantly attempting to re-key. This serves two purposes: first, this ensures forward secrecy by changing sharing Diffie-Hellman secret s, and consequently the keys generated from it used for encryption and authentication, and second, it provides the foundation for deniability. In each data message sent by each party there is a new proposed exponent $g^{x_{i+1}}$. Once received, the other party will encrypt its future messages with $g^{x_{i+1}y_i}$ and send its new key $g^{y_{i+1}}$.

This system has two useful properties: first, the parties can securely forget old Diffie-Hellman keys as soon as they are not required. Once again, this ensures forward secrecy, since any successful attack in the future will not be able to recover old keys and thus will not be able to decrypt the conversation. Second, constant re-keying contaminates the impact, that is, a successful attack will only allow an attacker to read a small portion of the conversation. Additionally, re-keying makes it possible to publish old MAC keys associated with old DH keys.

1.6 Socialist Millionaires' Protocol

Socialist Millionaires' Protocol (SMP) is a cryptographic protocol which allows two parties, Alice and Bob, that have secret information x and y respectively, to check whether $x = y$ without disclosing any additional information about x or y to a third party. If the values observed during SMP proof match for both

of the parties, then each party knows that the other party knows the same secret, so they can be sure about identities of each other, given the secret x hasn't been leaked.

As above, all exponentiations are done modulo p_{dh} and g_1 is generator of $\mathbb{Z}_{p_{dh}}^{\times}$. Assuming that Alice is the one initiating SMP:

- Alice:
  - Picks random exponents a_2 and a_3
  - Sends Bob $g_{2a} = g_1^{a_2}$ and $g_{3a} = g_1^{a_3}$
- Bob:
  - Picks random exponents b_2 and b_3
  - Computes $g_{2b} = g_1^{b_2}$ and $g_{3b} = g_1^{b_3}$
  - Computes $g_2 = g_{2a}^{b_2}$ and $g_3 = g_{3a}^{b_3}$
  - Picks random exponent r
  - Computes $P_b = g_3^r$ and $Q_b = g_1^r g_2^y$
  - Sends Alice g_{2b}, g_{3b}, P_b and Q_b
- Alice:
  - Computes $g_2 = g_{2b}^{a_2}$ and $g_3 = g_{3b}^{a_3}$
  - Picks random exponent s
  - Computes $P_a = g_3^s$ and $Q_a = g_1^s g_2^x$
  - Computes $R_a = (Q_a \times {Q_b}^{-1})^{a_3}$
  - Sends Bob P_a, Q_a and R_a
- Bob:
  - Computes $R_b = (Q_a \times {Q_b}^{-1})^{b_3}$
  - Computes $R_{ab} = R_a^{b_3}$
  - Checks whether $R_{ab} = (P_a \times P_b^{-1})$
  - Sends Alice R_b
- Alice:
  - Computes $R_{ab} = R_b^{a_3}$
  - Checks whether $R_{ab} = (P_a \times P_b^{-1})$

Source:[5]

If everything is done correctly, then R_{ab} should contain $(P_a \times P_b^{-1}) \times (g_2^{a_3 b_3})^{(x-y)}$. Which means that the verification will only succeed if $x = y$. Furthermore, since $g_2^{a_3 b_3}$ is a random number not known to any party, if $x \neq y$, no additional information is revealed.

SMP is used in OTR to detect impersonation and it may benefit from the changes being introduced to OTR. However, improvements to it are out of the scope of this thesis.

1.7 Protocol state machine

An OTR client maintains separate a state for each correspondent. The state consists of two variables: *message state* (`msgstate`) and *authentication state* (`authstate`).

1.7.1 Message state variable

The message state variable, `msgstate`, controls what happens to outgoing messages typed by the user[5]. It can take the following values:

State name	Description
`MSGSTATE_PLAINTEXT`	Initial state, messages are sent without encryption. This is the state that is used before an OTR conversation is initiated. *Note:* the only way to subsequently enter this state is for the user to explicitly request to do so via some user-initiated action.
`MSGSTATE_ENCRYPTED`	Outgoing messages are sent encrypted. This is the state that is used during an OTR conversation. The only way to enter this state is for the authentication state machine (below) to successfully complete.
`MSGSTATE_FINISHED`	Outgoing messages are not delivered at all. This state is entered only when the other party indicates she has terminated her side of the OTR conversation. The purpose of this state is to prevent accidentally leaking any information in plaintext.

1.7.2 Authentication state variable

The authentication state variable, `authstate`, shows the progress of the initial AKE. It can take the following values:

State name	Description
AUTHSTATE_NONE	Initial state.
AUTHSTATE_AWAITING_DHKEY	DH commit message was sent and DH key message is being expected from the other party.
AUTHSTATE_AWAITING_REVEALSIG	DH commit message was received and DH key message was sent in response. Reveal signature message is being expected.
AUTHSTATE_AWAITING_SIG	DH key message was received and reveal signature was sent in response. Signature message is being expected.
AUTHSTATE_V1_SETUP	For OTR version 1 compatibility.

If any of the parties in AUTHSTATE_AWAITING_SIG or AUTHSTATE_AWAITING_REVEALSIG, receives signature message or reveal signature message (and replies with her own signature message), respectively, then msgstate should transition to MSGSTATE_ENCRYPTED. Regardless of whether the signature verifications succeed, the authstate variable is reset to AUTHSTATE_NONE.[5]

1.7.3 OTR policies

OTR clients can choose to adhere to different *policies*. Policy, in the context of OTR state machine, is essentially a binary flag controlling the behaviour of the state machine during transitions. Policies can be set on global or per-correspondent basis.

Policy name	Description
ALLOW_V1	Allow version 1 of OTR protocol
ALLOW_V2	Allow version 2 of OTR protocol
ALLOW_V3	Allow version 3 of OTR protocol
REQUIRE_ENCRYPTION	Refuse to send unencrypted messages
SEND_WHITESPACE_TAG	Show support for whitespace tags
WHITESPACE_START_AKE	Start AKE after receiving whitespace tag
ERROR_START_AKE	Start AKE after receiving error message

1.7.4 State transitions

There are 12 events to which an OTR client must react:

Received messages:

Plaintext message without whitespace tag

Plaintext message with whitespace tag

Query message

Error message

DH commit message

DH key message

Reveal signature message

Signature message

Data message

User actions:

User requests to start an OTR conversation

User requests to end an OTR conversation

User types a message to be sent

The following description assumes that at least one of `ALLOW_V1`, `ALLOW_V2` or `ALLOW_V3` is set; otherwise, the OTR is disabled completely.

1.7.4.1 Receiving plaintext message without the whitespace tag

`msgstate` = `MSGSTATE_PLAINTEXT`:

Simply display the message to the user. If `REQUIRE_ENCRYPTION` is set, warn her that the message was received unencrypted.

`msgstate` = `MSGSTATE_ENCRYPTED` $\vee$ `MSGSTATE_FINISHED`:

Display the message to the user, but warn her that the message was received unencrypted.

1.7.4.2 Receiving plaintext message with the whitespace tag

`msgstate` = `MSGSTATE_PLAINTEXT`:

Remove the whitespace tag and display the message to the user. If `REQUIRE_ENCRYPTION` is set, warn her that the message was received unencrypted.

`msgstate` = `MSGSTATE_ENCRYPTED` $\vee$ `MSGSTATE_FINISHED`:

Remove the whitespace tag and display the message to the user, but warn her that the message was received unencrypted.

In any event, if `WHITESPACE_START_AKE` is set:

If tag offers OTR version 3 and `ALLOW_V3` is set:

Send a version 3 DH commit message, and transition `authstate` to `AUTHSTATE_AWAITING_DHKEY`.

Otherwise, if the tag offers OTR version 2 and `ALLOW_V2` is set:

Send a version 2 DH commit message, and transition `authstate` to `AUTHSTATE_AWAITING_DHKEY`.

1.7.4.3 Receiving a query message

If the query message offers OTR version 3 and `ALLOW_V3` is set:

Send a version 3 DH commit message, and transition `authstate` to `AUTHSTATE_AWAITING_DHKEY`.

Otherwise, if the message offers OTR version 2 and `ALLOW_V2` is set:

Send a version 2 DH commit message, and transition `authstate` to `AUTHSTATE_AWAITING_DHKEY`.

1.7.4.4 Receiving an error message

Display the message to the user. If `ERROR_START_AKE` is set, reply with a query message.

1.7.4.5 User requests to start an OTR conversation

Send an OTR query message to the correspondent.

1.7.4.6 Receiving a DH commit message

If the message is version 2 and `ALLOW_V2` is not set, ignore this message. Similarly if the message is version 3 and `ALLOW_V3` is not set, ignore the message. Otherwise:

`authstate` = `AUTHSTATE_NONE`:

Reply with a DH key message,
and transition `authstate` to `AUTHSTATE_AWAITING_REVEALSIG`.

`authstate` = `AUTHSTATE_AWAITING_DHKEY`:

This is the trickiest transition in the whole protocol. It indicates that the user has already sent a DH commit message to the correspondent, but that she either didn't receive it, or just didn't receive it yet, and has sent you one as well. The symmetry will be broken by comparing the hashed g^x you sent in the sent DH commit message with the received one, considered as 32-byte unsigned big-endian values.

If the sent message has is the higher hash value:

Ignore the incoming DH commit message, but resend the original DH commit message.

Otherwise:

Forget the original g^x value that was sent (encrypted) earlier, and pretend the state is `AUTHSTATE_NONE`;
i.e. reply with a DH key message, and transition `authstate` to `AUTHSTATE_AWAITING_REVEALSIG`.

`authstate` = `AUTHSTATE_AWAITING_REVEALSIG`:

Retransmit the original DH key message (the same one as was sent when the user entered `AUTHSTATE_AWAITING_REVEALSIG`). Forget the old DH commit message, and use this new one instead.

`authstate` = `AUTHSTATE_AWAITING_SIG` $\vee$ `AUTHSTATE_V1_SETUP`:

Reply with a new DH key message, and transition `authstate` to `AUTHSTATE_AWAITING_REVEALSIG`.

1.7.4.7 Receiving a DH key message

If the message is version 2 and `ALLOW_V2` is not set, ignore this message. Similarly if the message is version 3 and `ALLOW_V3` is not set, ignore this message. Otherwise:

`authstate` = `AUTHSTATE_AWAITING_DHKEY`:

Reply with a reveal signature message and transition `authstate` to `AUTHSTATE_AWAITING_SIG`.

`authstate` = `AUTHSTATE_AWAITING_SIG`:

If this DH key message is the same as the one received earlier (when entering `AUTHSTATE_AWAITING_SIG`):

Retransmit the reveal signature message.

Otherwise:

Ignore the message.

`authstate` = `AUTHSTATE_NONE` $\vee$ `AUTHSTATE_AWAITING_REVEALSIG` $\vee$ `AUTHSTATE_V1_SETUP`:

Ignore the message.

1.7.4.8 Receiving a reveal signature message

If the message is version 2 and `ALLOW_V2` is not set, ignore this message. Similarly if the message is version 3 and `ALLOW_V3` is not set, ignore the message. Otherwise:

`authstate` = `AUTHSTATE_AWAITING_REVEALSIG`:

Use the received value of r to decrypt the value of g^x received in the DH commit message, and verify the hash therein. Decrypt the encrypted signature, and verify the signature and the MACs.

If everything checks out:

Reply with a signature message.

Transition `authstate` to `AUTHSTATE_NONE`.

Transition `msgstate` to `MSGSTATE_ENCRYPTED`.

Otherwise:

Ignore the message.

`authstate` = `AUTHSTATE_NONE` ∨ `AUTHSTATE_AWAITING_DHKEY` ∨ `AUTHSTATE_AWAITING_SIG` ∨ `AUTHSTATE_V1_SETUP`:

Ignore the message.

1.7.4.9 Receiving a signature message

If the message is version 2 and `ALLOW_V2` is not set, ignore this message. Similarly if the message is version 3 and `ALLOW_V3` is not set, ignore the message. Otherwise:

`authstate` = `AUTHSTATE_AWAITING_SIG`:

Decrypt the encrypted signature, and verify the signature and the MACs. If everything checks out:

Transition `authstate` to `AUTHSTATE_NONE`.

Transition `msgstate` to `MSGSTATE_ENCRYPTED`.

Otherwise, ignore the message.

`authstate` = `AUTHSTATE_NONE` ∨ `AUTHSTATE_AWAITING_DHKEY` ∨ `AUTHSTATE_AWAITING_REVEALSIG`:

Ignore the message.

1.7.4.10 User types a message to be sent

`msgstate` = `MSGSTATE_PLAINTEXT`:

If `REQUIRE_ENCRYPTION` is set:

Store the plaintext message for possible retransmission, and send a query message.

Otherwise:

If `SEND_WHITESPACE_TAG` is set, and you have not received a plaintext message from this correspondent since last entering `MSGSTATE_PLAINTEXT`, attach the whitespace tag to the message. Send the (possibly modified) message as plaintext.

`msgstate` = `MSGSTATE_ENCRYPTED`:

Encrypt the message, and send it as a data message. Store the plaintext message for possible retransmission.

`msgstate` = `MSGSTATE_FINISHED`:

Inform the user that the message cannot be sent at this time. Store the plaintext message for possible retransmission.

1.7.4.11 Receiving a data message

`msgstate` = `MSGSTATE_ENCRYPTED`:

Verify the information (MAC, keyids, ctr value, etc.) in the message.

If the verification succeeds:

Decrypt the message and display the human-readable part (if non-empty) to the user.

Update the DH encryption keys, if necessary.

If you have not sent a message to this correspondent in some (configurable) time, send a "heartbeat" message, consisting of a data message encoding an empty plaintext. The heartbeat message should have the `IGNORE_UNREADABLE` flag set.

If the received message contains a Tag/Length/Value (TLV) element of type 1, forget all encryption keys for this correspondent, and transition `msgstate` to `MSGSTATE_FINISHED`.

Otherwise, inform the user that an unreadable encrypted message was received, and reply with an error message.

`msgstate` = `MSGSTATE_PLAINTEXT` $\vee$ `MSGSTATE_FINISHED`:

Inform the user that an unreadable encrypted message was received, and reply with an error message.

1.7.4.12 User requests to end an OTR conversation

`msgstate` = `MSGSTATE_PLAINTEXT`:

Do nothing.

`msgstate` = `MSGSTATE_ENCRYPTED`:

Send a data message, encoding a message with an empty human-readable part, and TLV type 1.
Transition `msgstate` to `MSGSTATE_PLAINTEXT`.

`msgstate` = `MSGSTATE_FINISHED`:

Transition `msgstate` to `MSGSTATE_PLAINTEXT`.

1.8 Previous work on OTR

There are a few papers about previous versions of OTR and its security analysis:

- Secure Off-the-record messaging[8]
- Finite-state security analysis of OTR version 2[1]

First paper shows vulnerabilities of the OTR version 1 and proposes using other well-known protocols for AKE, including SIGMA[6]. However, the attacks shown in the latter paper are applicable to the current version of the OTR and they affect the integrity and strong deniability of OTR under the assumption that the attack has complete control over the network. Attacks described there include:

- Version rollback — attacker can change the set of protocol versions supported by each party, forcing parties to use old OTR version 1. Fix is trivial: each party has to state what her initial version preference was at the end of AKE.
- Attack on strong deniability — dishonest party can reveal invalid MACs or not reveal them at all. This implies that the other party does not have appropriate information to forge message signatures, and thus can't deny contents of the messages. Fix of this attack is non-trivial.
- Attack on message integrity — attacker, due to assumed total network control, can hide the fact that one of the parties revealed her MAC keys, and subsequently present the other party a fake message signed with presumably valid MAC key. Fix of this attack is non-trivial.

- Authentication failure — attacker can attempt to perform a man-in-the-middle attack during AKE, leading to authentication failure, but only for one party. Due to the nature of AKE, the initiating party will believe that she successfully completed AKE, however, since the attacker didn't learn either party's Diffie-Hellman secret, no damage to confidentiality or integrity is done. Fix of this attack is to add information about whom the party believes she is communicating with, both encrypted and MAC'd in the final two AKE messages.

Nevertheless, mitigations for all of the aforementioned attacks are out of the scope of this thesis, because even with mitigations applied, as stated by papers' authors, "[one] *cannot rule out the possibility of additional attacks on the protocol*".

Chapter 2

Analysis and design

This section gives a brief description of what needs to be done to break OTR message confidentiality and how computationally complex the task is. Also, the improvements are proposed here.

2.1 Feasibility of computational attack

This section deals with a computational attack on the OTR message confidentiality. The attack scenario outlines how and what will be attacked and the complexity estimate follows in the next section along with mitigations.

2.1.1 Attack scenario

An attacker, who has a capability to passively observe all communication between Alice and Bob and wants to get the content of the messages, has to get the keys used for AES encryption. Those keys are derived from the shared DH key by hashing it in various ways. Thus, the attacker's main focus is recovering shared key s which was generated during initial AKE and the subsequent DH keys, which were present in the data messages. The attack is basically a classical discrete logarithm problem. Widely known classical algorithm for this is GNFS and it will be used in complexity estimation.

2.1.2 GNFS algorithm

General number field sieve algorithm is the most efficient classical algorithm[9] for factoring integers larger than 10^{100}. Heuristically, its complexity for factoring integer n is (using L-notation) of the form

$L_n\left[\frac{1}{3}, \sqrt[3]{\frac{64}{9}}\right]$

The algorithm consists of four computational stages, three of which depend only on the particular prime p (in our case p_{dh}). The first three stages also comprise most of the computation.

The brief description of each stage with possible improvements, as per Weak DH paper[12], is given below.

The first phase is *polynomial selection*, where a polynomial $f(z)$ defining a number field $\mathbb{Q}(z)/f(z)$ is searched for. It is parallelizable and takes only a small part of the time.

The second phase is *sieving*, factoring of ranges of integers and number field elements in batches to find many relations of elements, all prime factors of which are less than some bound B (called B-smooth). *special-q* lattice sieving may be used, which for each special q explores a sieving region 2^{2I} candidates, where I is a parameter. Such sieving parallelizes well, since each q is processed independently. Sieving is computationally demanding since it has to search through and try to factor many elements. Complexity of this step depends on heuristic estimate of the probability of encountering B-smooth numbers and on number I and on number of special q to consider before having enough relations.

The third phase is *linear algebra* stage, where a large, sparse matrix consisting of coefficient vectors of prime factorizations is constructed. A nonzero kernel vector of the matrix modulo the order q of the group produces logarithms of many small elements. This database of logarithms serves as input for the next stage. Computational cost depends on q and the matrix size and the stage can be parallelized up to some degree.

The last stage, *descent*, actually finds the logarithm of the given target. Sieving is repeated until the set of relations that allows to express the target logarithm in terms of logarithms in the database is found. It is done in three steps: initialization step, where attempts to express the target logarithm using medium-sized primes are made. Second step, where the primes from the previous stages are further sieved until they can be represented by elements in the database of logarithms. And the last step, that actually recovers the target using the logarithm database.

Recent work in number theory field has improved the descent stage complexity first to $L_n\left[\frac{1}{3}, 1.442\right]$[10] and later to $L_n\left[\frac{1}{3}, 1.232\right]$[11], which is orders of magnitude less that the previous stages[12].

2.1.3 Estimated complexity

In the WeakDH paper [12], authors have estimated that calculating the logarithm database for a 1024-bit prime would take 45M core-years and the descent stage would take only 30 core-days. Using the same approach as in the paper gives us estimated multiplicative factors by which the time and space complexity will increase. Time complexity of 1536-bit GNFS precomputation will increase by 6.048×10^5 and space complexity will increase by 777 with respect to the 1024-bit GNFS.

2.1.4 Recommendations on the DH group size from standards

One of the most respected cryptography standards, NIST SP 800-57 Part 1, states (revision 4, section 5.6.1)[4]: "... algorithm/key-size combinations that have been estimated at a maximum security strength of less than 112 bits are no longer approved for applying cryptographic protection on Federal government information".

In the same section, there is a table of strengths of different algorithm/key-size combinations. 2048-bit Diffie-Hellman has a rating of 112 bits there. Naturally, the 1536-bit Diffie-Hellman key exchange would have less than 112 bits of security making it improper for use in U.S. federal systems.

The table below is provided for reference.

1. Column 1 indicates the estimated maximum security strength (in bits) provided by the algorithms and key sizes in a particular row.
2. Column 2 identifies the symmetric-key algorithms that can provide security strength indicated in column 1.
3. Column 3 indicates the minimum size of the parameters associated with the standards that use finite-field cryptography (FFC). Examples of such algorithms include DSA and Diffie-Hellman. L is the public key size, and N is the private key size.
4. Column 4 indicates the value k (the size of modulus n) for algorithms based on integer-factorization cryptography (IFC). The predominant algorithm of this type is RSA. The value k is commonly considered to be the key size.
5. Column 5 indicates the range of f (the size of n, where n is the order of the base point G) for algorithms based on elliptic-curve cryptography (ECC). The value of f is commonly considered to be the key size.

Security Strength	Symmetric key algorithms	FFC	IFC	ECC
≤ 80	2TDEA	$L = 1024$ $N = 160$	$k = 1024$	$f = 160 - 223$
112	3TDEA	$L = 2048$ $N = 224$	$k = 2048$	$f = 224 - 255$
128	AES-128	$L = 3072$ $N = 256$	$k = 3072$	$f = 256 - 383$
192	AES-192	$L = 7680$ $N = 384$	$k = 7680$	$f = 384 - 511$
256	AES-256	$L = 15360$ $N = 512$	$k = 15360$	$f = 512+$

2.2 Proposed improvements

Since just changing the used algorithms to the new ones will not be beneficial in the long-term, changes shall consist of adding a flexible mechanism for selecting and using any cryptoprimitives equivalent to the ones used in OTR version 3, and providing support for additional Diffie-Hellman groups mentioned in IETF RFC 3526[7] and 5114[13]. Implementing these changes would require:

- A protocol negotiation mechanism
- A new key derivation scheme
- Changes to query and DH commit message contents
- Changes to low-level structure of all messages

2.2.1 Protocol negotiation mechanism

Before discussing the details of protocol negotiation, let's recapitulate what cryptoprimitives one needs during OTR AKE and data exchange:

Cryptoprimitive	**Algorithm used in OTR version 3**
Counter mode block cipher	AES with key length of 128 bits
Message authentication code	SHA1-HMAC and SHA256-HMAC-160 (first 160 bits of SHA256-HMAC output)
Multiplicative group of integers modulo n	1536-bit MODP Group[7]
Digital signature	Digital Signature Algorithm
Hash function	SHA256 and SHA-1

Special note about key derivation in OTR: SHA256 and SHA-1 are also used for deriving MAC and encryption keys. However, the way they are used puts unnecessary restrictions on key sizes, which is undesirable, and on the other hand, attempts to somehow extend the way of generating keys by using these functions may introduce additional vulnerabilities to the crucial parts of OTR. Thus, a key derivation function with variable length output has to be used.

Definition 1 *A cryptosuite is a collection of algorithms realizing aforementioned cryptoprimitives to be used during AKE and data exchange.*

When Alice and Bob want to start an OTR conversation, they need to negotiate which cryptosuite to use. They may do so by embedding information

about desired cryptosuites and their ranks, which are just positive integers chosen by each party independently, into message contents. Then they choose the cryptosuite with the highest average rank among the mutually supported cryptosuites and proceed to the later stages of the AKE.

2.2.2 Key derivation

During OTR conversation, parties regularly need to rekey, and consequently derive new keys. For AKE, six keys are needed ($c, c', m_1, m_1', m_2, m_2'$), and four more for data exchange. All of them are generated by hashing shared DH secret concatenated with some specific byte values. Due to the fact that MAC keys for data exchange (the ones that shall be revealed at some point) are generated by hashing of the encryption key, byte values used for them are changed. Otherwise identical keys will be produced both for encryption and authentication, and the latter will later be revealed, thus compromising the confidentiality of the conversation.

During the AKE, parties compute six keys based on the shared DH secret s:

- Two 128-bit AES keys, c, c'
- Four 256-bit SHA256-HMAC keys, m_1, m_1', m_2, m_2'

This is done in the following way:

- Encode the shared DH secret s as a byte sequence (concrete byte-level representation depends on the specific group used). Let this value be called *secbytes*.
- For a given byte b, define $h_2(b)$ to be a 256-bit output of SHA256 hash of the byte b followed by *secbytes* — $h_2(b) = \texttt{SHA256}(b||secbytes)$.
- Let c be the first 128 bits of $h_2(1)$ and let c' be the second 128 bits of $h_2(1)$.
- $m_1 = h_2(2)$
- $m_2 = h_2(3)$
- $m_1' = h_2(4)$
- $m_2' = h_2(5)$

During a data exchange, the parties use the most recent Diffie-Hellman keys to compute s and derive four keys based on it:

- Two 128-bit AES keys
- Two 160-bit SHA1-HMAC keys

They are calculated as follows:

- Each party determines if it is on the "low" or "high" end of the conversation. They do so by comparing their serialized public keys as a big-endian integer. This comparison has to be performed after every re-keying.
- A party sets two values, `sendbyte` and `recvbyte`, according to the results of the comparison: if the party is "high", then `sendbyte` = 1 and `recvbyte` = 2. Otherwise, the values are swapped.
- For a given byte b, define $h_1(b)$ to be 160-bit output of SHA-1 hash of the byte b followed by *secbytes* — $h_1(b) = \texttt{SHA1}(b||secbytes)$.
- The sending AES key is the first 128 bits of $h_1(\texttt{sendbyte})$
- The sending MAC key is the 160-bit SHA-1 hash of the 128-bit sending AES key.
- The receiving AES key is the first 128 bits of $h_1(\texttt{recvbyte})$
- The receiving MAC key is the 160-bit SHA-1 hash of the 128-bit receiving AES key.

As one may notice, such a key derivation scheme puts restrictions on key sizes, and consequently on available ciphers. In order to bypass this limitation, a flexible and secure key derivation scheme has to be used. Let us define $kd(b, l)$ as l-bit output of key derivation function of *byte sequence* b followed by *secbytes* — $(b||secbytes)$. Value l has to be adjusted according to each cipher's parameters.

With all aforementioned changes in mind, the AKE keys are derived this way:

- c is the first l bits of $kd(1, 2l)$
- c' is the last l bits of $kd(1, 2l)$
- $m_1 = kd(2, l)$
- $m_2 = kd(3, l)$
- $m_1{}' = kd(4, l)$
- $m_2{}' = kd(5, l)$

Data exchange keys are derived similarly:

- Each party determines which end of the conversation it is on, but with a subtle difference: if the party is "high" end, `sendbyte` = 6 and `recvbyte` = 7, and the reverse is true if the party is on the "low" end.

- Let $concat(a, b)$ be a byte sequence of byte a followed by byte b — $concat(a, b) = a||b$.
- The sending encryption key is $kd(concat(\texttt{sendbyte}, 1), l)$
- The sending MAC key is $kd(concat(\texttt{sendbyte}, 2), l)$
- The receiving encryption key is $kd(concat(\texttt{recvbyte}, 1), l)$
- The receiving MAC key is $kd(concat(\texttt{recvbyte}, 2), l)$

2.2.3 Changes to message contents

The structure of AKE demands that a cryptosuite must be determined for *both* parties at the time of DH key message creation. However, there is no need to send any extra messages before the regular AKE. The party sending the query message now has to embed the serialized list of supported cryptosuites. The other party, having received this message, now knows everything needed for cryptosuite selection and proceeds with DH commit message, which is now also modified to have an additional field holding a cryptosuite list. Thus, the party which is supposed to send the DH key message will successfully determine the cryptosuite, and proceed with the rest of AKE. Presence of cryptosuite list without any authentication data opens up a possibility of cryptosuite downgrade via man-in-the-middle attack. However, this problem can't be easily solved by simply signing the cryptosuite list, since it would be vulnerable to replay attacks — any attacker, who intercepted a signed cryptosuite list would be able to retransmit it, and it would still appear as a valid query message. A rather simple solution would be if each party selects her supported cryptosuites according to their security requirements i.e. the parties should not assume that query messages are somehow authenticated.

2.2.3.1 Unencoded messages

This section describes the messages in the OTR protocol that are not base-64 encoded binary.

OTR query message

This is the first message sent by the initiating party to indicate the beginning of an OTR conversation.

It consists of ASCII string `"?OTR"` followed by the supported versions of protocols and `"?"` at the end. If OTR version one is supported, then `"?"` is appended after `"OTR"`, otherwise `"v"` is appended.

Implementation supplied alongside this thesis accepts only `"?OTRvx?"` query message. The "`x`" stands for experimental.

2.2.3.2 Binary messages

This section describes the byte-level format of the base-64 encoded binary OTR messages. The binary form of each of the messages is described below. To transmit one of these messages, construct the ASCII string consisting of the five bytes `"?OTR:"`, followed by the base-64 encoding of the binary form of the message, followed by the byte `"."`.

Note: only messages whose structure had to be modified will be described there. All other messages are described in the OTR specification[5]. However, due to the variety of cryptographic algorithms used, implementations should use different field types: all field types other than `BYTE`, `SHORT` and `INT` have to be treated as `DATA` fields.

OTR DH commit message

This is the first message of AKE. It is sent to commit the choice of the DH encryption key while not revealing the key itself. It is a binary message and consists of several fields:

- Protocol version
- Message type
- Sender instance tag (ignored in the implementation)
- Receiver instance tag (ignored in the implementation)
- Encrypted DH key g^x
- Hash of g^x
- Cryptosuite list — serialized list of elements consisting of two integers: first denoting the cryptosuite ID, and the second denoting its rank.

OTR DH key message

This is the second message of the AKE. It simply consists of party's Diffie-Hellman key.

- Protocol version
- Message type
- Sender instance tag (ignored in the implementation)
- Receiver instance tag (ignored in the implementation)
- DH key g^y

OTR Reveal Signature message

This is the third message of the AKE. It reveals the key sent with the DH commit message and authenticates the party and the channel parameters. E is a counter mode cipher.

- Protocol version
- Message type
- Sender instance tag (ignored in the implementation)
- Receiver instance tag (ignored in the implementation)
- Revealed key r
- Encrypted signature:
  - $M_B = MAC_{K_{m_1}}(g^x, g^y, pub_B, keyid_B)$
  - $X_B = (pub_B, keyid_B, sig_B(M_B))$
  - result is $E_c(X_B)$
- MAC'd signature — $MAC_{K_{m_2}}(E_c(X_B))$

OTR Signature message

This is the final message of the AKE. The other party now authenticates herself and the parameters, thus finalizing the key exchange.

- Protocol version
- Message type
- Sender instance tag (ignored in the implementation)
- Receiver instance tag (ignored in the implementation)
- Encrypted signature:
  - $M_A = MAC_{K_{m_1}'}(g^y, g^x, pub_A, keyid_A)$
  - $X_A = (pub_A, keyid_A, sig_A(M_A))$
  - result is $E_{c'}(X_A)$
- MAC'd signature — $MAC_{K_{m_2}'}(E_{c'}(X_A))$

OTR Data message

This message is used to transmit a private message to the correspondent. It is also used to reveal old MAC keys.

- Protocol version
- Message type
- Sender instance tag (ignored in the implementation)
- Receiver instance tag (ignored in the implementation)
- Flags — bitwise OR of the flags for the message. Usually set to 0. Ignored in the implementation.
- Sender *keyid* — must be strictly greater than 0, and increment by 1 with each key change.
- Recipient *keyid* — must therefore be strictly greater than 0, as the receiver has no key with id 0. The sender and recipient keyids are those used to encrypt and MAC this message.
- DH y — next DH public key for this sender.
- Top half of counter block — this should monotonically increase (as a big-endian value) for each message sent with the same (sender *keyid*, recipient *keyid*) pair, and must not be all zeroes.
- Encrypted message — counter-mode encryption of the message using the appropriate encryption key derived from the sender's and recipient's DH public keys (with the keyids given in this message). Let initial counter be an x bytes long value whose first $x/2$ bytes are the above "top half of counter block" value, and whose last $x/2$ bytes are all zeroes.
- Authenticator — MAC of everything from the protocol version to the end of encrypted message using appropriate MAC key.
- Old MAC keys — discarded MAC keys.

Chapter 3

Library design

This section describes the design of the library, implementing aforementioned changes to the OTR protocol. The library is written in Python 3 and, depends on a few third-party libraries.

3.1 Main classes and points of interaction

The main point of interaction with the whole library is the `OTRSession` class. This class combines all the parts required for using OTR:

- `OTRStateMachine` class — handles all the logic
- `OTRSessionContext` class — contains all session data
- Public and private keys of correspondents
- A cryptosuite list
- A policy list

To construct an `OTRSession` object, one needs to pass his keypair and cryptosuite list, correspondent's public key, and optionally, a policy. In case of no selected policy, `REQUIRE_ENCRYPTION` will be selected by default. `OTRSession` contains a queue where all outgoing messages shall be put. Method `get_message` tries to get one message out of the queue, and if the queue is empty, it blocks until some other thread puts a new message in the queue. The message itself is an ASCII string. The client may send it in any desired form over any channel. Method `recv_raw` shall be called whenever client receives a new message, the message itself being passed as an argument. The return value of `recv_raw` shall be inspected for any errors or warnings. In case of an incoming data message, it will also contain the decrypted plaintext message. When a client wants to send a message, `send_text` needs to be called, with plaintext message as the argument. A client must start an OTR session before sending any

text messages. It must be done by either calling `start_session` (if the client initiates first) or receiving a query message and calling `recv_raw`. Ending the session is done in a similar manner, by calling `end_session`.

3.2 Cryptosuites

The main goal of this library is to provide a set of interfaces and expected behaviors required for implementing all the cryptoprimitives in OTR. This way, when anybody wants to use some new algorithms with OTR, the only work required will be implementing classes realizing those interfaces.

3.2.1 AbstractCryptoSuite class

Classes inheriting from `AbstractCryptoSuite` should contain references to all cryptoalgorithms used during an OTR session. This class should not have any operations defined over it. Each class inheriting from `AbstractCryptoSuite` class should be given some integer identifier and reference to it shall be included in a global variable where all subclasses of `AbstractCryptoSuite` shall reside.

3.2.2 AbstractCryptoObject class

All objects on which operations will be performed (DH/MAC/cipher keys, etc.) shall inherit from this class. The default constructor has to initialize the whole internal state to some reasonable default. The instances shall implement a few public methods:

- `generate_random` - fill appropriate internal variables with random data. Cryptographically secure randomness sources have to be used, though the exact choice is implementation-defined.
- `pack` - pack the whole internal state of the object into a `bytes` array.
- `unpack` - do the opposite of `pack`.

3.2.3 Cryptoproviders

Cryptoproviders are the classes realizing the actual cryptographic operations. They operate on the aforementioned objects. There are two main types of cryptoproviders: stateful and stateless. Stateful ones are the CTR ciphers and the MAC providers. Stateless ones are hash, signature, Diffie-Hellman and KDF providers. Each provider has to inherit from its specific class.

3.2.3.1 CTR cipher provider

- `set_key/get_key` - set/get key for this instance.
- `set_counter/get_counter` - set/get counter block for this instance.
- `encrypt` - perform encryption on a given `bytes` object. *Note*: During each encrypt call, the top half of the counter block should remain the same, while the bottom half should be filled with zeroes before encryption begins.
- `increment_top_half` - increment the top half of the counter block as a big-endian integer.

3.2.3.2 MAC provider

- `set_key/get_key` - set/get key for this instance.
- `mac` - produce MAC of the given `bytes` array.

3.2.3.3 Hash provider

- `hash` - produce a hash of the given `bytes` array.

3.2.3.4 Modular group provider

- `generator` - return a generator of this group.
- `modexp` - perform exponentiation modulo this group.

3.2.3.5 Signature provider

- `sign` - sign the given `bytes` array with the private key.
- `verify` - verify the given signature of the given `bytes` array against the given public key.

3.2.3.6 KDF provider

- `kd` - produce output of length *len* from the byte sequence consisting of byte *b* followed by *secbytes*.

3.3 Demonstration code

There is a demo code included with sources which demonstrates basic usage of library. It creates a short local OTR conversation and shows its every stage. An implementation of all cryptoproviders is provided as well. It uses AES-256, SHA-256, RSA-2048 as a signature with SHA-256 as a hash and Probabilistic

Signature Scheme (PSS) with MGF1 as a mask generating function, Diffie-Hellman over 2048 MODP group as per RFC 3526[7], and SHA256-HMAC as MAC and Argon2id as a KDF. Argon2id is used with default parameters provided by third-party library except the random salt length, which has to be equal to zero in order to make key derivation deterministic.

Testing code and cryptoproviders are dependent on the following python libraries:

- `argon2-cffi`
- `cryptography`

They can be installed by using tool called `pip` or manually.

Conclusion

The main goals of this thesis were a discussion of an attack on OTR protocol security, obstacles which complicate mitigation and proposition of a possible solution. The attack is based on property of the GNFS algorithm that allows to perform most of the heavy computational work in advance for a specific Diffie-Hellman group and thus makes solving individual instances of discrete logarithm much faster given the precomputed database. Since the amount of precomputational work required depends only on Diffie-Hellman group, and the security properties of OTR conversation depend on the shared Diffie-Hellman secret, even constant re-keying present in OTR is not helpful in mitigating this attack. Thus, support for additional and larger Diffie-Hellman groups is needed to raise the cost of precomputing beyond any reasonable budgets. The solution proposed in this thesis allows *any* cryptoalgorithm equivalent to the one used in original OTR to be used instead of the default ones. This solution aims to provide OTR ability to adapt and maintain its security properties as new attacks are discovered on some of the cryptoalgorithms. The solutions gives a significant flexibility for OTR, since now removing vulnerable algorithms from OTR or adding new ones is just a matter of implementing proper cryptoproviders. This makes it easy to greatly increase cost of computational attack by just choosing the right cryptosuites, just like the TLS protocol. A Python library implementing basic OTR facilities has been created as proof of concept. The design of library allows to quickly plug in any new algorithm by implementing just a handful of interfaces. Further work may consist of improving SMP in OTR, by adding support for additional groups just like in the case with Diffie-Hellman. Another area of improvement is asynchronous messaging. OTR, in its current state, requires both parties to be present during conversation; i.e. it is a synchronous messaging protocol. However, constant online presence might be problematic on mobile devices due to network conditions. Thus, a protocol aiming to become successful in that market needs some asynchronous messaging support.

Bibliography

[1] Bonneau, J.; Morrison, A. Finite-State Security Analysis of OTR Version 2.

[2] Borisov, N.; Goldberg, I.; Brewer, E. Off-the-record Communication, or, Why Not to Use PGP. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, New York, NY, USA: ACM, 2004, ISBN 1-58113-968-3, pp. 77–84, doi:10.1145/1029179.1029200. Available from: `http://doi.acm.org/10.1145/1029179.1029200`

[3] Marc Stevens, P. K. A. A. Y. M., Elie Bursztein. The first collision for full SHA-1.

[4] Barker, E. Recommendation for Key Management Part 1: General. Technical report, jan 2016, doi:10.6028/nist.sp.800-57pt1r4. Available from: `https://doi.org/10.6028/nist.sp.800-57pt1r4`

[5] Off-the-Record Messaging Protocol version 3. Available from: `https://otr.cypherpunks.ca/Protocol-v3-4.1.1.html`

[6] Krawczyk, H. *SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, ISBN 978-3-540-45146-4, pp. 400–425, doi:10.1007/978-3-540-45146-4_24. Available from: `https://doi.org/10.1007/978-3-540-45146-4_24`

[7] Kivinen, T.; Kojo, M. More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). RFC 3526, RFC Editor, May 2003, `http://www.rfc-editor.org/rfc/rfc3526.txt`. Available from: `http://www.rfc-editor.org/rfc/rfc3526.txt`

[8] Di Raimondo, M.; Gennaro, R.; Krawczyk, H. Secure Off-the-record Messaging. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, WPES '05, New York, NY, USA: ACM, 2005, ISBN

1-59593-228-3, pp. 81–89, doi:10.1145/1102199.1102216. Available from: http://doi.acm.org/10.1145/1102199.1102216

[9] Yang, L. T.; Huang, G.; Feng, J.; et al. Parallel GNFS algorithm integrated with parallel block Wiedemann algorithm for RSA security in cloud computing. *Information Sciences*, volume 387, 2017: pp. 254 – 265, ISSN 0020-0255, doi:https://doi.org/10.1016/j.ins.2016.10.017. Available from: http://www.sciencedirect.com/science/article/pii/S0020025516312348

[10] Commeine, A.; Semaev, I. An Algorithm to Solve the Discrete Logarithm Problem with the Number Field Sieve. In *Public Key Cryptography - PKC 2006*, edited by M. Yung; Y. Dodis; A. Kiayias; T. Malkin, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ISBN 978-3-540-33852-9, pp. 174–190.

[11] Thomé, E. *Algorithmes de calcul de logarithmes discrets dans les corps finis*. Theses, Ecole Polytechnique X, May 2003, membre du Jury : von zur Gathen, Joachim et Coppersmith, Don et Berger, Thierry et Villard, Gillles et Sendrier, Nicolas et Roblot, Xavier. Available from: https://pastel.archives-ouvertes.fr/tel-00007532

[12] Adrian, D.; Bhargavan, K.; Durumeric, Z.; et al. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[13] Lepinski, M.; Kent, S. Additional Diffie-Hellman Groups for Use with IETF Standards. RFC 5114, RFC Editor, January 2008.

Appendix A

Low-level details of OTR

A.1 Data message plaintext structure

Note: this describes only the human-readable part of the data message.

Each plaintext message (either before encryption, or after decryption) consists of a human-readable message (encoded in UTF-8 encoding), optionally followed by:

- a single byte with value of 0
- zero or more TLV (type/length/value) records with no padding between them

Each TLV takes the form:

Type — 2 bytes, big-endian integer

Length — 2 bytes, big-endian integer

Value — sequence of bytes

Type 1 TLV denotes the other party's request to finish an OTR conversation. The party sending it should also reveal all remaining MAC keys and transition the `msgstate` variable to `MSGSTATE_PLAINTEXT`. The party receiving it should also reveal the keys and transition `msgstate` to `MSGSTATE_FINISHED`.

In the implementation produced on the basis of this thesis, TLVs are not used at all. Instead, a message consisting of just a single byte 0 is treated as Type 1 TLV.

A.2 MAC keys revealing

Whenever one is about to forget either one of their own old DH keys, or one of the correspondent's old DH public keys, one should take all of the receiving

MAC keys that were generated from that key (note that there are up to two: the receiving MAC keys produced by the pairings of that key with each of two of the other side's keys; but note that one only needs to take MAC keys that were actually used to verify a MAC on a message), and put them (as a set of concatenated byte sequences) into the "Old MAC keys to be revealed" section of the next data message. This in done to allow the forgeability of OTR transcripts: once the MAC keys are revealed, anyone can modify an OTR message and still have it appear valid. But since we don't reveal the MAC keys until their corresponding DH keys are being discarded, there is no danger of accepting a message as valid which uses a MAC key which has already been revealed.

Appendix B

Acronyms

AES Advanced Encryption Standard.

AKE Authenticated Key Exchange.

HMAC Hash-based Message Authentication Code.

MAC Message Authentication Code.

OTR Off-The-Record messaging.

PKI Public Key Infrastructure.

SHA-1 Secure Hash Algorithm 1.

SMP Socialist Millionaires' Protocol.

Appendix C

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `poc` ..... implementation sources
    - `pyotr` ..... library sources
    - `test.py` ..... simple demo code
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format