O^2TR: Offline Off-the-Record (OTR) messaging

Mahdi Daghmehchi Firoozjaei, Sang Min Lee, and Hyoungshick Kim

Electrical and Computer Engineering Department,
Sungkyunkwan University, Suwon 440-746, Republic of Korea
{mdaghmechi,sangmin91,hyoung}@skku.edu
http://seclab.skku.edu/

Abstract. Off-the-record (OTR) is a security protocol that can be used in privacy preserving instant messaging (IM) systems. However, the conventional OTR is not applicable in some practical scenarios (e.g., when communication network became disconnected) because OTR requires both parties to be online at the same time. To address this limitation, we extend the conventional OTR into a new protocol named offline OTR (O^2TR). O^2TR makes the conversation parties be able to handle an offline message even when a session connection is lost. To show the feasibility of the proposed protocol, we implemented a prototype to support O^2TR based on the Gajim XMMP instant messaging platform. Our experiments showed that O^2TR can reliably be used when a network party is broken down. Moreover, O^2TR provides an efficient session refreshment which is about 34% faster than the original OTR.

Keywords: Key exchange, OTR, Instance messenger

1 Introduction

Off-the-record (OTR) is a security protocol to make a private session over an instant messaging (IM) system [1]. OTR provides end-to-end confidentiality and integrity of the transmitted messages by using encryption, authentication, perfect forward secrecy (PFS), and deniability services. In OTR systems, ephemeral keys are newly established for every session by updating key parameters continuously. These features make the OTR protocol a suitable solution for privacy preserving in the online services. By the deniability feature, no one, including the sender, can prove who authored a particular message in the transcript [2, 3]. On the other hand, PFS makes it impossible to compromise the past session keys even when long-term keys are compromised. Basically, OTR mimics the features of an actual face-to-face conversation into the virtual environment for deniability service [2]. Server-based security solutions establish private sessions for IM clients through a central server while OTR establishes secure sessions without relying on a central server; after establishing secure sessions, OTR provides a secure environment for pairwise encryption and authentication.

In general, asynchronous communication, where the sending and receiving of a message do not need to happen simultaneously, is a usual phenomenon in the

most IM systems. In the central server-based security protocols, by providing the confidentiality between client and server, there is no issue to handle offline messages. For other security mechanisms, such as OTR, which are not server-based, there is a different condition. To start a private session, OTR requires both conversation parties to be online in order to exchange their Diffie-Hellman (DH) parameters for the key exchange. Furthermore, the encryption and authentication keys in each OTR data packet are securely chained to the previous keys. Based on the spirit of OTR, this protocol is only applicable to synchronous communications and does not support the offline messages. Although this condition does not challenge OTR privacy provision, its practicality can be limited by this constraint.

To address the offline message limitation, we extend the original OTR into a new protocol named offline OTR (O^2TR). Basically, the original OTR is based on a mechanism of advertising keys and receiving confirmations for those keys in subsequent messages [4]. After establishing a private session, OTR parties exchange their next ephemeral keys in each data packet. In fact, the keys in OTR are chained together. The proposed O^2TR utilizes this feature to handle offline messages. Our experiments showed the feasibility of O^2TR to handle offline messages. Although processing time to handle O^2TR messages is similar to OTR's, O^2TR provides faster private session refreshment. The main contributions of this paper are summarized as follows:

- In O^2TR, we introduce a solution to detect and store the latest keys of the private session to handle offline messages. We develop a mechanism to renew and replace the stored keys based on the incoming and outgoing messages. By detecting an offline message, the stored keys of the previous session are used to handle the message.
- To evaluate the performance of our model in a real network environment, we implemented O^2TR on Gajim IM system launched on an XMPP/Jabber platform.

The rest of this paper is organized as follows. In Section 2, an overview of OTR and the offline message issue are presented. Section 3 provides a model explanation of O^2TR. The implementation of O^2TR and the evaluation results are respectively explained in Sections 4 and 5. In Section 6, we analyze the security of O^2TR and our conclusions are summarized in Section 7.

2 Overview of OTR

The protocol of OTR was initially introduced by Borisov et al. [1] in 2004. OTR is built on a high-level cryptographic abstraction, such that AES (128-bit), SHA-256 and SHA-1 hash functions are used in the last version of OTR (Ver. 3). SHA-256 is used in the handshake process, and to calculate the encryption and authentication keys based on a DH share. On the other hand, SHA-1 is used as an HMAC function to authenticate each encrypted message [5]. Therefore, all encrypted messages are authenticated by the HMAC function. Using encrypted

DH key exchange and keys derived from a shared secret to encrypt each message and to authenticate each ciphertext lead to building public key authenticated encryption on top of lower-level primitives [6].

To establish a private session, the conversation parties share their DH keys by OTR handshake protocol. A version of SIGMA is used as the authenticated key exchange (AKE) to exchange the generated keys. To prevent the man-in-the-middle (MitM) attack, no clear DH encryption key is revealed in the first AKE message. After exchanging the DH keys in the first two messages, the conversation parties authenticate each other by their public keys in an encrypted channel. These keys are long-lived DSA public keys and are used only for authentication [5][7]. After the authentication process, the private session is established and the OTR parties are ready to exchange data packets.

Figure 1 shows the format of an OTR data packet. In this packet, the header field shows the version of OTR protocol, message type, the sender and the receiver of the message, and the required flags. As mentioned before, OTR exploits the ephemeral keys for encryption to provide PFS service. Since the keys are changed per message, *keyid*s are used to select the latest DH keys and to make sure that a unique set of keys is being used [7]. In fact, a *keyid* is an integer number that is increased by generating a new DH key.

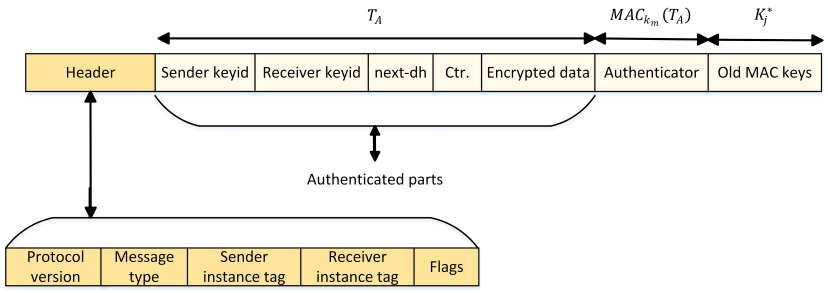


Fig. 1: OTR Data packet format

Since the encryption key is pairwise between two OTR parties, in each data packet a new DH public key is suggested by the sender for the next data message. The suggested key in the *next_dh* field will be used for the next message after receiving its confirmation. DH share is computed based on both parties' DH keys to generate the encryption and authentication keys. AES algorithm in CTR mode is used to encrypt/decrypt the messages. The *Encrypted data* field consists of the encrypted message. In the rest fields, the MAC value and the used MAC keys are revealed. By revealing the MAC keys one round later to the public, OTR provides message deniability [8]. Therefore, once the MAC keys are revealed, anyone can modify the message and the receiver cannot prove the sender's authorship.

2.1 OTR and offline messages

The offline message problem occurs when a conversation party in an OTR private session goes offline or leaves the IM chat room. Due to the structure of OTR to exploit the pairwise encryption, it is only applicable in the synchronous communications where both the communicating parties are online [9]. During an OTR conversation, if the receiver party gets offline any undelivered message remains in the IM server side as an offline message. The offline message will be delivered whenever the receiver shows up. Since the offline message is encrypted based on the last lost private session, it is unreadable for the receiver.

As shown in Fig. 1, in OTR data packet, the next ephemeral DH share (in the *next_dh* field) is protected with a MAC computed with the current key. Until being acknowledged by the recipient this new DH share cannot be used by the sender [9]. Therefore, to pursue the key generation chain in OTR, both conversation parties require to be online. This requirement makes OTR impracticable in the asynchronous communication.

In OTR, the authentication and the message states indicate the current state of each OTR conversation. The authentication state shows the authentication progress, while the message state indicates the ciphering state of the outgoing messages [5]. To prevent any flaw and inconsistency for sending messages, the message state controls what happens during the OTR conversation. For instance, if during a private session a party logs out his OTR client the correspondent will be notified. This notification prevents to unwillingly send any message in plaintext mode [5].

Since the liveness of an OTR private session depends on the running IM chat room, the message state mismatching occurs when a client leaves the chat room while its OTR session is not finished. In this case, the current OTR private session is not terminated on the other side. Therefore, any message sent by the correspondent is based on the previous unfinished private session. The unreadability error will happen when the client receives an offline message from the previous unfinished private session.

As an example of this mismatching, Fig. 2 shows the related OTR error message in a Pidgin-OTR messaging client. During an OTR conversation one party, $mdf3@localhost$, gets offline and leaves the chat room. After showing up and connecting back to the Pidgin IM server, an offline message that was sent by $mdf2@localhost$ and has remained on the server is delivered to $mdf3@localhost$. Since there is no private session and this OTR message was encrypted with the last session, it is not readable.

To prevent any OTR parameter mismatching in the asynchronous communication, we introduce O^{2}TR to handle offline messages. As stated earlier in the Section 2, by sending or receiving an OTR message, all keys are regenerated, based on the new DH shared key. Therefore, each party knows the required keys of the next message. By retaining these keys in O^{2}TR, we are able to retrieve any offline OTR message's required keys. In this case, an O^{2}TR party can authenticate and decrypt the offline message that was created in the last session.

Fig. 2: The error message for receiving an offline message in the Pidgin-OTR IM client

3 Offline OTR- O^{2}TR

In order to achieve PFS, OTR exploits the ephemeral keys to renew the encryption and authentication keys for every data packet. To this end, each OTR party generates a new DH key and shares it in the *next_dh* field of the data packet. Therefore, each party knows the next DH key of the correspondent. Based on this fact, saving the next DH key for every packet is the basic concept of O^{2}TR.

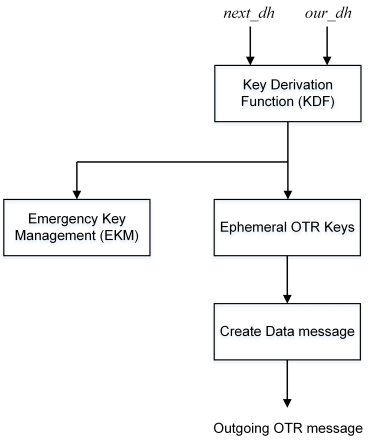


Fig. 3: Storing the latest keys in EKM

Basically, the O^{2}TR model consists of two extra parts in comparison to the original OTR; emergency key management (EKM) and offline message detection

(OMD). As shown in Fig. 3, all ephemeral keys for encryption and authentication generated by the key derivation function (KDF) are saved in EKM too. These keys are generated based on the DH keys of both parties. Since new keys are generated for each message, the registered keys in EKM are replaced with the new keys. Therefore, in each conversation EKM consists of the latest keys in both O^{2}TR parties. Unlike the ephemeral keys, EKM retains the registered keys even if the client leaves the chat room.

To handle offline messages, in O^{2}TR model, OMD checks the state of the incoming message. In the case of offline message, an encrypted message is delivered to a client with no private session. Therefore, the state of an offline message does not match with the self state of the receiver that is in the plaintext mode. As shown in Fig. 4, by detecting this mismatching by OMD, the message is considered as an offline message, and is handled by EKM stored keys instead of rejecting.

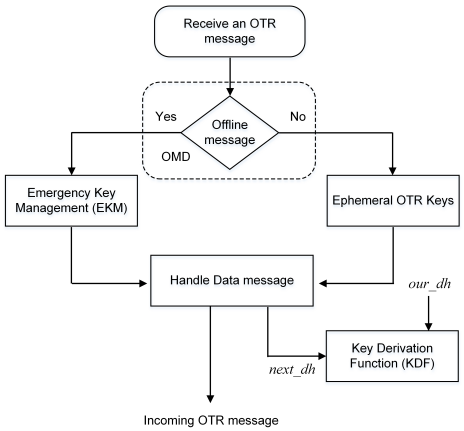


Fig. 4: Handling the offline message in O^{2}TR

Since the offline message is created by the last unfinished O^{2}TR session, no MAC keys are released and they are valid yet. Therefore, it is impossible to compromise this offline message's validity before releasing its MAC key in the next O^{2}TR data message [5].

4 Implementation

To analyze the performance of O^{2}TR, Gajim[1] messaging application was selected as an IM system. Furthermore, eJabberd XMPP[2] server, implemented in Ubuntu 14.04, was used to provide an XMPP/Jabber platform to launch the Gajim

[1] https://gajim.org/index.php?lang=en
[2] https://www.process-one.net/en/ejabberd/

IM. Due to the abilities of eJabberd to provide secure client-to-server (C2S) connections (i.e., SSL/TLS connection) and a storage for undelivered messages, this server was selected for our implementation. Figure 5 shows a typical IM connection through an eJabberd server.

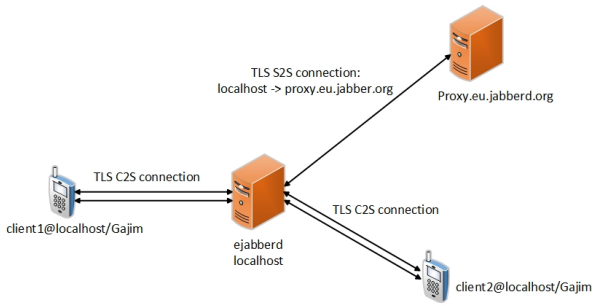


Fig. 5: IM connection through an eJabberd XMMP sever

To enable OTR for the Gajim XMPP/Jabber client, we used OTR plugin provided by Braden available on GitHub[3]. Furthermore, the Pure-Python-OTR, a Python OTR implementation package, available on GitHub[4] was installed to enable this OTR plugin. To implement O^{2}TR in the Gajim XMPP/Jabber client, we optimized the Pure-Python-OTR package and OTR plugin to add EKM and OMD modules.

5 Experiments

To evaluate and scrutinize the performance of O^{2}TR model, the concepts of offline message recovery and processing time for message handling were considered. These items were scrutinized on the IM communication between Gajim XMMP/Jabber clients in two models: OTR and O^{2}TR.

5.1 Offline message recovery

To evaluate the ability of O^{2}TR to recover the offline message, we considered a condition that led to disconnect a normal Gajim-OTR conversation. That event was carried out for more than 200 separate Gajim IM conversations. As the results, all offline messages (100%) were recovered and decrypted completely with Gajim-O^{2}TR while it was not possible for the normal Gajim-OTR model. Figure 6 shows the offline message delivery capability in Gajim-O^{2}TR IM conversation.

[3] https://github.com/python-otr/gajim-otr
[4] https://github.com/python-otr/pure-python-otr

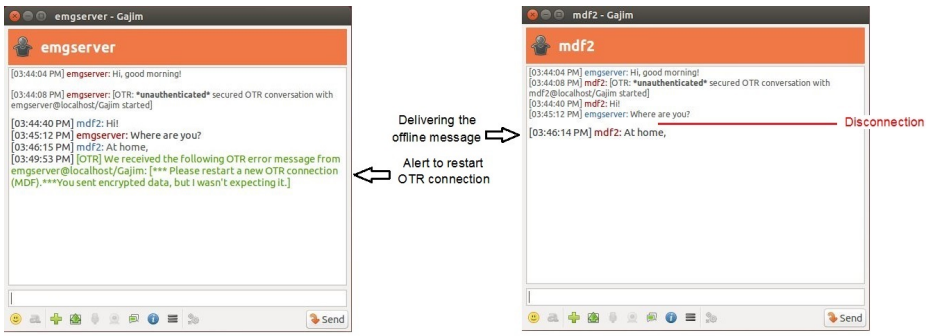
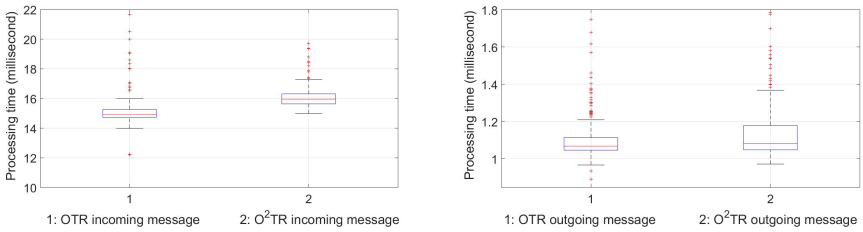


Fig. 6: Retrieving offline message in Gajim-O^{2}TR

5.2 Processing time

To provide a practical comparison, we evaluated the processing time of message exchange, handshake, and session refreshment in both protocols (O^{2}TR and OTR). As shown in Fig. 7, there are no meaningful differences between processing time to handle the incoming and outgoing messages in OTR and O^{2}TR protocols. Unlike sending outgoing messages, handling incoming messages takes more time due to performing some additional tasks, such as scrutinizing the current policies (accepting encrypted or non-encrypted messages), message type (data packet or AKE), OTR's parameters (version and validity), key IDs, and authenticate the received message.



(a) Incoming message (b) Outgoing message

Fig. 7: Processing time to handle the incoming message (a) and outgoing message (b) in OTR and O^{2}TR

Table 1 compares the average processing time to handle all messages in OTR and O^{2}TR protocols. The average processing time to handle incoming and outgoing messages is almost same in both protocols. Based on these results, O^{2}TR does not impose significant processing time to IM clients. To handle offline messages, which is only possible in O^{2}TR, averagely 29.95ms is needed for packet

processing. In comparison to incoming message handling in OTR, this amount is almost two times longer. It should be noted that this time is only spent for the first offline message in which the required keys are extracted from the local memory.

Table 1: Average processing time to handle incoming, outgoing, and offline message in OTR and O^{2}TR

Message type	OTR	O^{2}TR
Incoming message	15.15ms	16.07ms
Outgoing message	1.11ms	1.14ms
Offline message- first	—	29.95ms

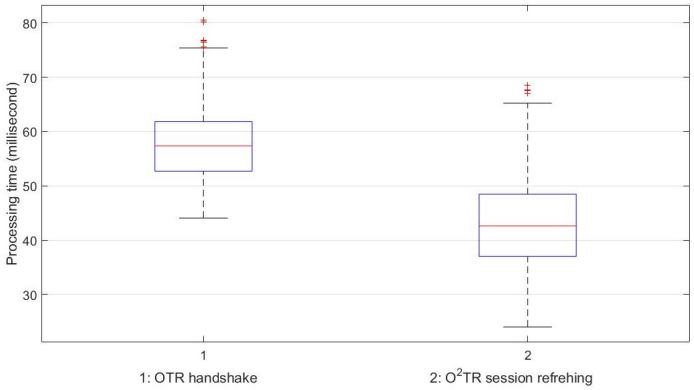


Fig. 8: Processing time for handshaking in OTR and session refreshment in O^{2}TR

On the other hand, O^{2}TR provides a condition to refresh the private session when an offline message is received. Therefore, the private session is refreshed with no interrupt instead of starting a new OTR handshake which is longer. Figure 8 shows the processing time for OTR handshaking and to refresh the private session in O^{2}TR. Averagely, an OTR handshake takes 58.29ms while the refreshment of a private session takes 43.49ms in O^{2}TR. Based on these results, in O^{2}TR a private session is refreshed 34% faster than creating a new

private session in OTR. Consequently, in a private session disconnection, O^{2}TR not only retrieves offline messages but also refreshes the interrupted private session faster.

6 Security analysis

As one of the basic security properties of OTR, PFS feature assures to protect session keys in which case leads to compromising server's private key. Based on the chain of key generation in OTR, each data packet consists of a new DH key suggested by the sender, which will be used for the next message. It's worth noting that, until the receiver has acknowledged this advertised next key, it can't be used. If an OTR party needs to send multiple messages before receiving any replies, he/she will need to keep using the current key and advertising the same next key [4]. This condition leads to a possible flaw when a curious IM server as an active attacker filters the packets and forces the client to send all messages with the same one key.

Since all packets are encrypted, the success of the attacker to detect the ciphering keys depends on the possibility of breaking AES-CTR protocol. To prevent any key leakage possibility, we define a threshold (e.g., $k = 5$) to use an encryption key to do multiple encryption rounds. Therefore, no O^{2}TR party can use the same encryption key more than k times for multiple messages. Otherwise, the sending is paused until receiving a reply or refreshing to a new private session.

On the other hand, since no ephemeral keys are reused, the property of PFS is not compromised. Based on this fact, the attacker, the curious server in our adversary model, has no chance to detect the private session key even by logging the revealed keys. Consequently, any kind of the replay attack is not practical to compromise O^{2}TR protocol as well as OTR. Finally, after retrieving the offline message the private session will be refreshed by the receiver. In this state, the same security level as OTR's is provided for the conversation parties.

7 Conclusions

We proposed O^{2}TR, an offline OTR protocol, that makes the conversation parties be able to handle offline messages. O^{2}TR utilizes EKM and OMD parts to save the latest keys, detect and retrieve the offline message even after leaving the chat room. To prevent any key leakage flaw, we set a threshold to use an encryption key to do multiple encryption rounds. Our experiments showed the complete capability to handle offline messages under different conditions. It's worth noting that, O^{2}TR does not impose significant processing time to IM clients. Furthermore, O^{2}TR provides an efficient session refreshment which is about 34% faster than creating a new private session in OTR.

Acknowledgments. This work was supported in part by the IITP (No. B0717-16-0116), the ITRC (IITP-2017-2015-0-00403) and the MISP (R2215-16-1005).

The authors would like to thank all the anonymous reviewers for their valuable feedback.

References

1. N. Borisov, I. Goldberg, and E. Brewer. 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, pages 77–84. ACM, 2004.
2. H. Liu, E. Y. Vasserman, and N. Hopper. Improved group off-the-record messaging. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 249–254. ACM, 2013.
3. M. Di Raimondo, R. Gennaro, and H. Krawczyk. Secure Off-the-record Messaging. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, WPES '05, pages 81–89. ACM, 2005.
4. M. Marlinspike. Advanced cryptographic ratcheting. https://whispersystems.org/blog/advanced-ratcheting/, 2013. Open Whisper Systems.
5. I. Goldberg, D. Goulet, and J. V. Bergen. Off-the-Record Messaging Protocol version 3. https://otr.cypherpunks.ca/Protocol-v3-4.1.1.html.
6. W. M. Petullo, X. Zhang, J. A. Solworth, D. J. Bernstein, and T. Lange. MinimaLT: Minimal-latency Networking Through Better Security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security*, CCS '13, pages 425–438. ACM, 2013.
7. C. Alexander and I. Goldberg. Improved user authentication in off-the-record messaging. In *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, WPES '07, pages 41–47. ACM, 2007.
8. I. Goldberg, B. Ustaoğlu, M. D. Van Gundy, and H. Chen. Multi-party Off-the-Record Messaging. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 358–368. ACM, 2009.
9. T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and Th. Holz. How secure is textsecure? Cryptology ePrint Archive, Report 2014/904, 2014. http://eprint.iacr.org/2014/904.