OMEMO: CRYPTOGRAPHIC ANALYSIS REPORT

For PACIFIC RESEARCH ALLIANCE

V1.0
Amsterdam
June 1st, 2016

Document Properties

Title	OMEMO: CRYPTOGRAPHIC ANALYSIS REPORT
Version	1.0
Author	Sebastian Verschoor
Reviewed by	Melanie Rieback
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	May 10th, 2016	Sebastian Verschoor	Initial draft
0.2	May 11th, 2016	Sebastian Verschoor	Conversations developer fixed the issue I found
0.3	June 1st, 2016	Sebastian Verschoor	Changed client organisation
1.0	June 1st, 2016	Sebastian Verschoor	Final version

Contact

For more information about this Document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Overdiemerweg 28 1111 PP Diemen The Netherlands
Phone	+31 6 10 21 32 40
Email	info@radicallyopensecurity.com

Table of Contents

1 Introduction

The OMEMO protocol is an adaptation of the Signal Protocol, created by Open Whisper Systems[1] . OMEMO is designed to work in an XMPP environment [26][28], where users can have multiple devices with which they want to communicate with each other. An XMPP session can involve multiple servers, instead of just one Open Whisper Systems server. The impact of multiple servers should be minimal, as a trusted server was never part of the security model that guarantees the security of the Signal Protocol.

The predominant part of this report, the protocol security analysis, can be found in Section 2, in which I analyze the full OMEMO protocol, including the used Signal protocol and the protocol for encrypted file transfer. Section 3 discusses the results of a brief inspection of the open-source code [7] of the Conversations application [9], as a reference implementation of the OMEMO specification. Finally, Section 4 provides a summary of results and my recommendations for the OMEMO standard.

1.1 Terminology

OMEMO is a recursive acronym that stands for “OMEMO Multi-End Message and Object Encryption”. In this report, the term OMEMO refers to the protocol as specified by its ProtoXEP [29], also called OMEMO version 0.

In order to eliminate confusion, Open Whisper Systems has very recently [20] changed the name of their protocol from the difficult to pronounce “Axolotl” to the “Signal Protocol”. The old name has been used to refer to both the entire protocol and to refer to just the ratchet component of the full protocol. The OMEMO specification was created before this announcement and uses the old names. This report follows the new terminology: “**Signal Protocol**” refers to the full protocol, “Triple Diffie-Hellman” refers to the initial handshake and “Double Ratchet” refers to the ratchet algorithm. I recommend that the OMEMO specification updates their terminology as well.

Throughout this report, I will follow the tradition in cryptographic literature of naming the end-users Alice and Bob, while reserving the name Eve to represent the adversary. Note that the end-users represent persons, not the device (or multiple devices) that they use.

1.2 Attacker Model

Section 2 of the OMEMO ProtoXEP lists only a few requirements for the protocol. From a cryptographic perspective, many basic requirements are missing, including the basic CIA triad[2] . That does not mean that the protocol does not meet those requirements, it just means that the specification is not as explicit as it can and should be. This section aims to extend the requirements to list all security properties that OMEMO achieves.

To claim that the protocol is secure, a well-defined attacker model is required in order to specify what the protocol is secure *against*. By defining the goals that adversaries might have and defining their capabilities, it

[1] https://whispersystems.org/
[2] Confidentiality, Integrity and Availability

becomes clear what the protocol needs to defend against and which security properties it should provide to the end-users.

1.2.1 Attacker Goals

The attacker goals are closely tied to the security properties of the secure messaging protocol. Table 1 lists the different goals that an attacker might have and the corresponding security property that a protocol should provide in order to be considered secure.

Table 1: Attacker Goals

Attacker Goal	Security property
Compromise messages	Confidentiality of messages
Alter sent messages	Integrity of messages
Inject false messages	Authenticity of messages
Identify as another person	Authentication of communication partner
Block communication	Availability of communication
Learn communication metadata	Privacy protection
Prove *what* was said	Deniability of message content
Prove that two persons communicated	Deniability of the conversation
Learn past communication after compromise	Forward secrecy
Prolong a successful attack	Future secrecy

Not every attack can be defended against by a secure messaging protocol. It is especially hard to provide availability when an attacker is assumed to be able to block messages on the communications network. Having said that, the protocol should not make it easy for an attacker to block communication.

To protect the privacy of the users, the protocol should not leak metadata about the users' communication, such as who they are communicating with, how many messages they sent and from where. Communication layers below the secure messaging protocol might leak this data as well, but it could be hidden through anonymity tools such as Tor. In that case, the protocol itself should not reveal any metadata.

To provide deniability, it should be impossible for anyone to provide convincing proof to a third party about past communication. To deny that any conversation ever took place is a stronger claim than just denying the precise contents of a message.

Forward secrecy[3] and future secrecy are properties that ensure some damage control in case that a device or key does get compromised. Forward secrecy ensures that keys that are currently on the device do not compromise any past communication, so that the impact of a device compromise is minimized. Future secrecy

[3]also called Perfect Forward Secrecy or Key Erasure

ensures that an attacker that has compromised a key in the past, does not get to prolong his attack indefinitely. This is often achieved by introducing fresh randomness that should remain unknown to a passive adversary.

1.2.2 Attacker capabilities

A base model for the attacker is the Dolev-Yao model [5], in which the attacker has full control over the network. The attacker can listen to, alter, inject and drop any message on the network.

However, real attackers have capabilities beyond control over the network. By inspecting the physical properties of the implementation, they might learn secret information that is on the communication device. This is called a side-channel attack. Device compromises can also be achieved by low-tech attacks such as a rubber-hose attack or through legal procedures. An attacker is assumed to learn information through side-channels and to be able to get temporary access to the device.

An issue with some existing protocols is that users need to trust in the communications server that is being used. The open nature of XMPP allows arbitrary parties, including adversaries, to set up a fully functional XMPP server. But even if you trust the organization that runs the server, you might not trust the government of the country in which the server is located to protect your privacy. Therefore, the attacker is assumed to have full control over the server that is used for communication.

The last capability that is given to the attacker is to compromise protocol participants themselves. When Alice communicates with Bob, the protocol should provide some pro- tection in case Bob turns out to be a dishonest participant. Basically, the protocol should enforce Bob to play by the rules.

2 Protocol Analysis

The OMEMO standard is best described as a wrapper protocol around the Signal protocol. I will analyze the standard as specified by its ProtoXEP [29], in order to find if it achieves the cryptographic properties that it claims to uphold. In addition, this report analyses the OMEMO Encrypted Jingle File Transfer protocol as specified by its ProtoXEP [11].

In Section 2.1, I will first briefly inspect the Signal Protocol, to see how it achieves its security properties. Those already familiar with the Signal Protocol might want to skip this section. After that, Section 2.2 will fully analyze how the OMEMO protocol uses the secure sessions created by Signal to set up an OMEMO session between multiple devices of two users.

At the moment of writing, version 3 is that latest version of the Signal Protocol. This is the version that is used by OMEMO version 0 and the one that is analyzed in this report.

2.1 Signal Protocol

Although the Signal Protocol is mentioned in the specification, there is no reference given to this protocol.[4] This is not a flaw of the OMEMO specification, because a normative specification for the Signal Protocol does not exist. The open-source library that Open Whisper Systems provides on GitHub [30] is a straightforward implementation and I will use it as a basis for my analysis of the Signal Protocol. In addition to the source code, OWS published a series of blog posts [18][17][16] that further clarify how their protocol works.

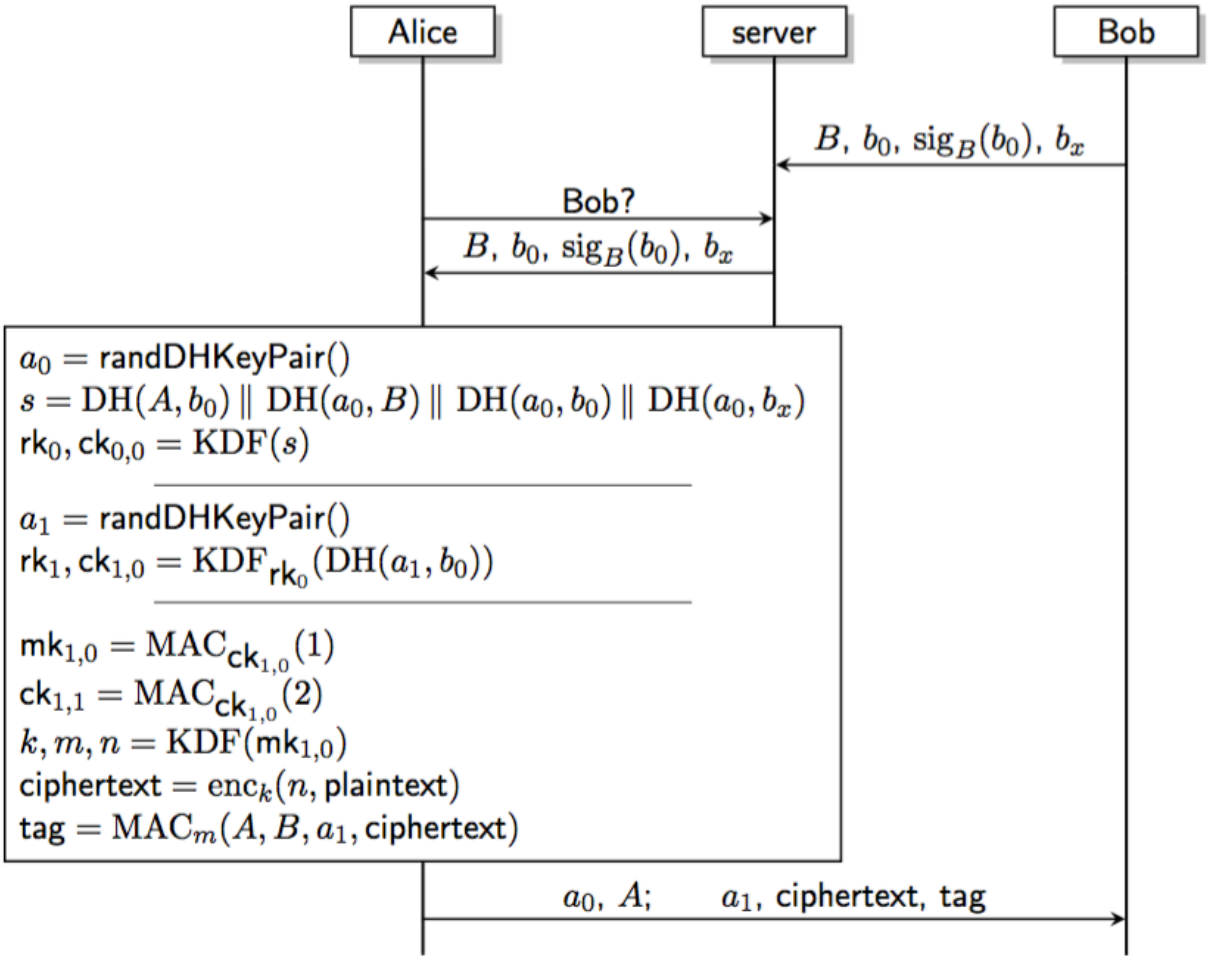

Figure 1: Signal Protocol version 3

2.1.1 Protocol Description

A simplified representation of the Signal Protocol is given in Figure 1. The figure shows the start of a conversation between Alice and Bob. In this abstracted example, the participants are identified by their name. In reality, this would be a phone number for the Signal application and an XMPP address in case of OMEMO.

[4]The OMEMO website [10] references to Trevor Sprain's GitHub page [24], but this is only a draft specifi- cation of the Double Ratchet part of the protocol.

Notation The following notation is used: $KDF_s(i)$ derives a key using salt s, info data i and a constant label that is unique for each KDF computation in the figure. When no salt is specified, the constant value 0 is used. $MAC_k(m)$ computes an authentication tag on message m, using key k. $enc_k(n, m)$ computes the symmetric encryption of message m, using key k and nonce/initialization vector n. To keep the diagram simple, the precise meaning for asymmetric keys notation depends on the context, but it is straightforward. For example: a_0 refers to the entire key pair when generated, to the private key when used in the DH computation and to the public key when sent in the message. Only public keys are sent in messages.

Prekeys First Bob uploads his client-side generated key material to the server so that he can be contacted by Alice. He sends his long-term identity key B, his signed prekey b_0 with corresponding signature $sig_B(b_0)$ and a one-time-use prekey b_x. Bob can go offline at this point, the server will now act as an online cache for others that want to initiate a conversation with Bob.

TripleDH When Alice wants to talk with Bob, she requests the cached data from the server. The server complies and Alice can initiate the TripleDH[5] handshake. She first generates her own one-time key pair a_0. She combines the keys by concatenating the results of the DH computations and computes s, a shared secret that initializes the Double Ratchet. Using the KDF function, Alice computes the initial root key rk_0.

DH ratchet (every reply) Alice updates the root key with the DH ratchet. She first generates a fresh random key pair a_1 and does a DH computation with the latest DH key she received from Bob (initially b_0). Using the previous root key rk_0 as a seed for the KDF, she computes a new root key rk_1 and a new sending chain key $ck_{1,0}$. At this point, Alice should delete the old root key rk_0 and her previous key pair a_0 to ensure forward secrecy.

Chain ratchet (every message) Alice derives a message key ($mk_{1,0}$) and a new chain key $ck_{1,1}$ from the old chain key $ck_{1,0}$ and she deletes the old chain key for forward secrecy. Alice derives three keys from the mk with the KDF: an encryption key k, an authentication key m and a nonce/initialization vector n. She encrypts the plaintext message and computes an authentication tag over the (public) identity keys and the ciphertext. She then sends the SignalMessage to Bob, consisting of her one-time key a_1, the ciphertext and the authentication tag. Only with the PreKeySignalMessage (the first message) will she also include her first one-time key a_0 and her identity key A. Bob can use the key material from the PreKeySignalMessage to initiate the root ratchet and receiving chain ratchet, from which the key material can be derived to validate and decrypt the message.

This diagram implicitly also shows how the conversation continues. Every time the user replies to a message, the steps below the first horizontal line are taken: the root key is updated with a fresh random DH computation and a new sending chain ratchet is initialized. For every additional message, the sending chain key is updated and a fresh message key is used to encrypt user messages. Note that both users have one root ratchet and two chain ratches: one for sending and one for receiving.

Key verification In order to ensure that no man-in-the-middle attack has taken place, Alice needs to verify that the identity key she has connected with indeed belongs to Bob. How they do this is not important, as long as it happens over an authenticated channel, but no PKI is assumed in the protocol. Instead, users must manually verify the identity key “fingerprint” (which is just the full public key) of the other party.

Message counters Messages might arrive out of order and can even arrive after the DH ratchet has been forwarded. Therefore, the sender of the message also includes two counters: one for how many messages were sent under the current ratchet and one for the total under the previous ratchet. With these counters, the

[5] In older versions, the key derivation did indeed consist of three DH computations: it did not include the signed prekey. The name “QuadrupleDH” is not used to avoid confusion with a variant that also includes a DH computation between the identity keys. The current computation can be referred to as a variant on standard TripleDH: “TripleDH with signed and one-time prekeys”.

receiver can see exactly which messages did not (yet) arrive and store only the corresponding message key mk. These counters are authenticated by the tag, but they are not encrypted.

Multiple prekeys In a real-world situation, Bob would want more than one person to be able to communicate with him, so he uploads multiple prekeys to the server. In the case of the Signal application, Alice only gets a single one-time prekey from the server. When the server runs out of prekeys, Alice can complete the handshake without Bob's one-time prekey. This message has reduced forward secrecy, because Bob cannot delete the signed prekey b0 immediately after use. When Bob receives a PreKeySignalMessage, he should send a fresh signed prekey to the the server, so that the key that is cached on the server gets updated.

Bob needs to know which signed prekey and which one-time prekey Alice used in her computation, so each prekey has its own identifying number. Alice includes that number in the PreKeySignalMessage and sends Bob, unauthenticated and unencrypted. These numbers are generated sequentially.

Key lifetimes The identity key lasts indefinitely. It is possible that Alice sends a message using a signed prekey that was already updated by Bob. For that reason, Bob should keep a few old signed prekeys in storage, so that he does not need to discard those messages. How long this should be is not specified, but the specification should include at least a guideline and/or upper bound for this lifetime. The one-time prekeys are used only once and should be deleted immediately after use. The server should delete a public one-time prekey immediately after they handed it out to someone, so it does not get used again. DH ratchet keys should be deleted after the other party has sent their next DH ratchet key and that DH computation has been completed.

Used cryptographic primitives The protocol so far is lacking a description of which cryptographic primitives are used as building blocks of the protocol. Technically, the protocol does not need to be locked, but at this moment it is non-trivial to change the used ciphers in the OWS code. The following primitives are in use:

- enc: AES [23] in CBC mode and using PKCS5padding
- MAC: HmacSHA256 [14]
- KDF: HKDF [15] using HmacSHA256
- DH: X25519 [1]
- sig: Ed25519 [2]

A standard of the protocol could benefit from allowing different primitives or cipher suites. For example, when a cryptographic breakthrough leads to breakage of a primitive, clients can simply reject all suites that use that primitive and remain secure. Or an implementer might want to use a different suite because of business requirements or performance issues. This cipher suite should be negotiated at the start of the protocol: Bob can upload a list of all suites he accepts to the server cache and Alice can pick one. To avoid downgrade attacks, the full list and the picked suite should be authenticated in the PreKeySignalMessage.

Note that the identity key B is used both for signing prekeys and in a DH computation, which is secure [4] with the current implementation over Curve25519, but might not be trivial to implement for other public key ciphers. The used structure of encrypt-then-MAC could also be replaced with an authenticated encryption cipher/mode as long as it allows for additional authenticated data (AAD).

Metadata The protocol leaks metadata about who is communicating with whom and how much they are communicating. Alice's request for the server cache leaks to the server that she wants to start a conversation with Bob, as does the PreKeySignalMessage. The plaintext message counters that are included in each SignalMessage make it possible to track the rest of the conversation.

Unlike the ratchet used in the Signal Protocol, the regular variant of the Double Ratchet [24] also encrypts the message headers, which would make it possible to avoid tracking of the conversation. It would only make sense to implement this if this information is not leaked already in the transport layer.

2.1.2 Security Analysis

A more thorough analysis of the Signal protocol has been done before by Frosch, Mainka, Bader, Bergsma, Schwenk and Holz [6]. In their work, the researchers provide a detailed description of the application, the underlying protocol and the environment in which the application operates. That environment includes the Google Cloud Messaging infrastructure in order to send push messages to the devices.

In their analysis, the researchers found no major weaknesses in the Signal Protocol. They give security proofs for the building blocks that make up the Signal Protocol: the initial key exchange, the subsequent key derivation and the authenticated encryption. In addition, they identify a minor weakness in the authentication of users identity keys, named the unknown key-share attack, and they comment on the claimed additional security features (future secrecy, forward secrecy and deniability).

Unknown key-share attack In an unknown key-share attack, Eve downloads the public key material of Bob and uploads the keys as if they are her own. When Alice wants to initiate a conversation with Eve, she checks that the identity key she downloaded from the server match with the one that Eve presents to her out-of-band. Alice completes the handshake on her side and sends here initial messages. Eve forwards these (still encrypted) messages to Bob.[6]

The result of a successful attack is that Alice falsely believes that she sent her messages to Eve, while Bob falsely believes that the received messages were intended for him. Eve is unable to compromise the confidentiality or integrity of the messages, making the impact of this attack relatively low.

The underlying cause of the above attack is that Eve never needed to prove to Alice that she was in possession of the private key corresponding to the presented identity public key. The researchers propose a solution, where the users engage in an out-of-band interactive zero-knowledge proof over an authenticated channel, such as exchange of messages with QR-codes. Because this solution is based on an interactive protocol, it would disable users from sending messages immediately if the recipient is not online at that moment.

Future secrecy Future secrecy ensures that a key compromise at some point in time will not propagate indefinitely. The Signal protocol achieves this by introducing new randomness with every reply in order to forward the root ratchet. A key compromise by a passive attacker will not propagate from that point on. However, an active attacker that has compromised both the root key and an identity key is able to set up a man in the middle attack that can be prolonged indefinitely.

Forward secrecy Forward secrecy ensures that when a device is compromised, no past messages can be decrypted. This is achieved by erasing message encryption/decryption keys as soon as possible. One of the problems with the Signal Protocol is that Bob's private prekeys need to remain stored on the device until a message has been received that was encrypted with the corresponding public prekey. If Eve manages to intercept and block that message from being delivered, Bob will keep holding on to that private prekey, so that Eve can read the content of the message if she is able to extract Bob's private prekeys from his device. But for

[6]Forwarding messages is trivial for an attacker, because we assume she has full control of the server.

any message that is delivered and decrypted correctly, Bob discards the private part of the prekey and ensures forward secrecy.

Version 2 of the Signal Protocol was also vulnerable to an attack on the forward secrecy of the first message by an active adversary. Eve could provide her own prekey (of which she knew the corresponding private key) and provide it to Alice, pretending it was the prekey of Bob, together with Bob's identity key. Bob would not be able to decrypt the message, but Eve would be able to if she was able to compromise just Bob's private identity key. Version 3 fixes this vulnerability by introducing adding a prekey that is signed by the identity key. This signature ensures that Eve cannot provide her own prekey and pretend that it belongs to Bob, thus preventing the attack.

Deniability Deniability for a messaging application can occur on two levels: denial of the message content and denial of the full conversation. The researchers prove that the Signal Protocol achieves the former, but they claim that the latter might only be theoretical. Because clients authenticate to the Open Whisper Systems server (similar to how an XMPP client authenticates to an XMPP server) and this server needs to know the addresses of the sender and recipient in order to guarantee delivery, the logs that might be stored by the server can reveal that a conversation took place.

The fact that a conversation took place might leak, but through another layer than the application layer of the core Signal Protocol. The solution to such leaking of metadata should also be contained in the appropriate layer and should stay out of scope for the OMEMO specification.

2.2 OMEMO

OMEMO uses Signal in order to set up a session. In Section 2.2.1, I will show how OMEMO uses those Signal sessions in order to set up a secure conversation between multiple devices. In Section 2.2.2, I will analyze the cryptographic strength of the design and describe minor issues I found in the specification. Two major problems are described in their own sections: Section 2.2.3 explains how a malicious device can compromise the entire conversation and Section 2.2.4 shows how forward secrecy and future secrecy can be affected by other devices.

2.2.1 Protocol description

At a very high level, OMEMO works similar to how a Signal group messages [19] work, but with multiple devices instead of multiple users. A Signal session is set up between each device. Messages are encrypted and authenticated with a random key and the encryption of that key is sent as message content of a SignalMessage.

A complete overview of OMEMO is given in the use cases of section 4 of the ProtoXEP, but I will provide a brief description here. A typical XMPP setup is shown in Figure 2. Alice is registered at a different server as Bob. Alice has registered two OMEMO enabled devices, while Bob has only registered his phone and wants to register his laptop as well.

In order to register his laptop, Bob generates a random 31-bit device id and registers it by adding it to his device list on the server via PEP. He then generates a random identity key B, a signed prekey b_0 with corresponding signature sig(b_0) and 100 one-time prekeys b_x. He then uploads this in an OMEMO bundle, again via PEP. This bundle contains the same information that Bob caches on the server in regular Signal.

Assume Alice wants to send an OMEMO encrypted message from her phone. She can detect that Bob's device(s) support OMEMO by requesting his device list with PEP. If he does, she encrypts and authenticates her message using a randomly generated key. For every device that Alice wants to send the encrypted message to, she fetches the entire bundle via PEP. If she wants to add more of her own devices in the conversation, she gets their bundles as well from her own server. Alice creates a PreKeySignalMessage for every device by picking a random one-time prekey from each bundle and encrypting the randomly generated key to each device. She combines all information in a single MessageElement: the encrypted payload (`<payload/>`), the plaintext iv (`<iv/>`), the sender id (`sid`) and the encrypted random key (`<key/>`) tagged with the corresponding receiver id (`rid`).

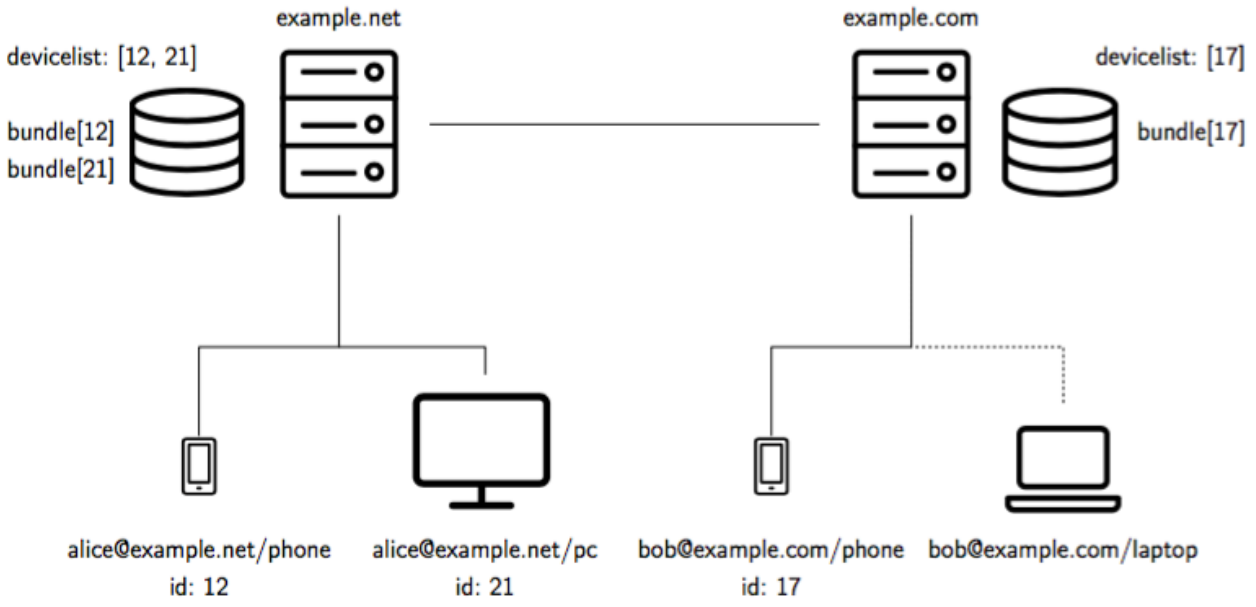

Figure 2: OMEMO version 0

Bob's device can decrypt the message by selecting the correct `<key/>` element based on the rid attribute and use it to initialize the Signal session on his side.

At this point, Alice's phone has set up a Signal session with each of the devices. If Bob wants to reply, he still needs to initialize a session with Alice's PC, so he also needs to download all bundles and initialize Signal sessions by sending a PreKeySignalMessage where necessary. If all devices (but one) have sent a message, each device will have a pairwise Signal session set up.

Device synchronization The regular delivery mechanism of XMPP was built to send a message to one user only and to send it only to online devices. Message Carbons [13] are used to deliver the messages to multiple devices per user and Message Archive Management (MAM) [21] is used to enable delivery to devices that are currently offline. This achieves inter-client history synchronization if no malicious device is taking part in the conversation.[7]

[7] see also Section 2.2.3.

The MAM was designed as a message archive, but instead it is used here as a message cache. The ciphertext messages will remain stored online after they have been downloaded, even though the keys will be discarded upon encryption. This does not affect security, but it wastes space on the server. A client should delete the message from the server after they decrypted it and deleted the message keys.

KeyTransportElement Instead of sending a MessageElement, a device can also send a message without a payload, called a KeyTransportElement. The randomly generated key might be used for example to encrypt a file, see Section 2.3. Sending a KeyTransportElement also has the advantage that the Signal ratchet gets forwarded.

Prekey collision When Alice wants to create a PreKeySignalMessage for Bob, she gets the full bundle and randomly selects one of his prekeys. When Bob receives multiple PreKeySig- nalMessages, the prekeys might collide. Because of the birthday problem, collisions are expected to occur often. With 100 prekeys a collision is expected after 12.3 PreKeySignalMessages and for the suggested minimum of 20 keys, a collision is expected after approximately 5.86 PreKeySignalMessages.

When Bob receives PreKeySignalMessages with prekey collisions, he replies to Alice with a KeyTransportElement containing his own PreKeySignalMessage, so that a new session can be initiated. If Bob no longer has the corresponding private prekey, he silently discards the message.

When fetching a PreKeySignalMessage with MAM, Bob should keep the private prekey in memory (but he may also delete them) until all MAM messages have been downloaded, so that he can still decrypt messages. He can decrypt, but he should set up a new session with Alice anyway. The specification warns for a small subgroup attack [22] that applies when reusing one-time keys. However, that attack does not apply to X25519 [3]. Implementers should make sure that the prekeys also get discarded if the MAM catch-up does not complete successfully (for example when the device crashes), or the forward secrecy of the message will be compromised.

A more elegant solution would be to do what OWS does: let the server send each one-time prekey once and delete them afterwards, instead of delivering the entire list of prekeys. That way, no collisions can occur on the prekeys and fewer initial messages get dropped. When the server runs out of one-time prekeys, the server lets Alice know and she can complete the PreKeySignalMessage without a one-time key, just as the Signal application.

It is unclear if this solution is possible to implement in XMPP, as it appears that there currently is no XMPP extension that allows a server to delete/mark PEP nodes while the user is offline.

Device ID The resourcepart of the XMPP address [27] is not used, but instead a separate device id is used. This is because the resourcepart can change during an OMEMO session, in which case a device will no longer be able to detect the correct key in the header. With the current setup, the device id should be unique among all device ids that participate in a conversation, so they potentially collide with any other device in use. Using 31 random bits for a device id might be enough to avoid a collision most of the time, but if the full XMPP address were used instead the user can guarantee no collisions as he only needs to take care of not colliding with himself.

Colliding device ids do not affect the security of the protocol: in the worst case, colliding devices are unable to participate in the conversation, affecting only the usability.

2.2.2 Security Analysis

The pairwise Signal session in OMEMO are very similar to that of the Signal application, so their security properties are similar. The server model for XMPP is slightly different as that of OWS, but since the protocol does not rely on trust in the server this should not affect the security of the Signal sessions. The way that multiple Signal sessions are combined to create a multi-device OMEMO session does affect the security properties of the entire protocol, so I will analyze that in this Section.

Signed prekey lifetime OMEMO does not specify when a signed prekey should be renewed on the server. When this key does not get updated, the forward secrecy of a PreKeySig- nalMessage is not protected against an active attacker (see Section 2.1.2). The device should send a fresh key to the server regularly and old signed prekeys should be deleted from the device after a while.

Cryptographic primitives OMEMO adds only one cryptographic primitive: authenticated encryption of the payload, which is fixed to AES in GCM mode. There is no reason to fix the cipher for OMEMO, any form of encryption with authentication can be used. A non-authenticated encryption cipher can also be used when the payload authentication is included in the tag of the SignalMessage, as described in Section 2.2.3.

The specification should allow for alternative ciphers, for the same reason that the Signal protocol should. Preferably, the negotiation of this cipher should be merged with that of the negotiation of the Signal cipher suite, so that clients only need to negotiate this once at the start of a conversation. Unfortunately, Signal is not standardized and it would probably be unwise to specify in the OMEMO standard how Signal should negotiate its primitives.

Metadata Communication metadata is already leaked through the Signal protocol and probably also through the XMPP transport layer, but OMEMO also leaks this information through the plaintext device ids. The payload is encrypted in GCM mode, so the size of the plaintext is also leaked.

2.2.3 Malicious device

One cannot expect messages to remain confidential when one of the participating devices is malicious. However, a user might suspect at least that the integrity of messages sent by an honest device is guaranteed by the protocol. After all, a secure Signal session with that honest device has been set up. However, the Signal session only protects the random key. A malicious device has access to that key and can thus re-encrypt and re-authenticate any payload with that key, without the receiving party being able to detect it. This is illustrated in Figure 3.

The displayed attack only shows the attack in one direction: Eve is able to modify and read anything sent by Alice. Eve needs to apply the same attack to Bob in order to setup up a bidirectional man in the middle attack. Note that Eve needs to strip of her own `<key/>` element from the list of keys in every message in order to remain undetected from Bob.

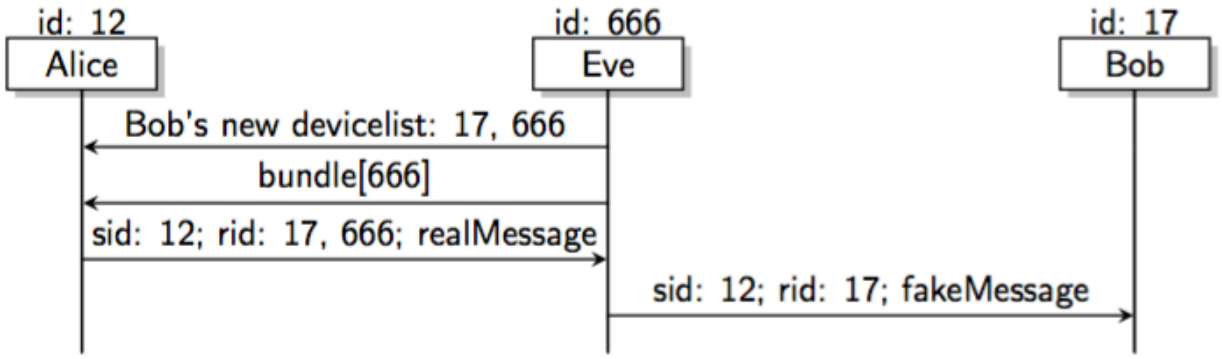

Figure 3: OMEMO man in the middle attack

Two careful users will not be susceptible to this attack, because neither of them will ever accept an unvalidated key. However, no matter how careful Bob is with validating the identity key of the sending device, he must assume that Alice has never made a mistake and none of the devices were compromised in order to be guaranteed the authenticity of messages that come from any of her devices. This trust in the other party is not necessary, if the messages were authenticated inside the Signal session. Also, Bob could make it less likely for Alice to accept a malicious device by creating a cryptographic link between devices.

Message authentication Messages are authenticated by the randomized key, which protects the message integrity from outsiders. However, anyone with access to the key can alter the message, which includes a malicious device. There are a few possible mitigations, each with their advantages and disadvantages.

A possible solution would be to authenticate inside the Signal session. By authenticating the payload with the tag of the SignalMessage, the full message is authenticated in such a way that no other device can compromise the integrity. The ciphertext (and not the plaintext) of the payload message should be authenticated, so that the MAC-then-encrypt pattern is applied.[8] This solution increases the computational load on the sender side, because the payload needs to be authenticated more than once. When the ciphertext is added as authenticated additional data (AAD) of the Signal message, it would reduce the message size slightly, because no authentication tag is required on the payload. The payload encryption method should then be simplified to a non-authenticated block cipher mode. It will also require some alterations on the Signal library, as the current implementation does not allow the library user to add their own AAD.

The payload can also be authenticated by including a hash of the payload ciphertext in the SignalMessage plaintext (and therefore the corresponding encrypted hash in the SignalMes- sage ciphertext). This would not require changes to the Signal library, but it would increase the size of each `<key/>` element. This solution is less elegant than the previous, as the hash of the payload ciphertext is sent encrypted, even though the recipient can compute this value themselves.

By authenticating a list of all recipient device ids in the tag of the SignalMessage, Bob has a guarantee about which devices Alice has sent the message to. Bob's client might provide him with a warning if that list includes untrusted devices. This protects him against the specific attack described above, but the protocol remains vulnerable if one of the devices gets compromised by another attack. This solution can be combined with the above solution of authenticating the payload ciphertext with the SignalMessage ciphertext or tag.

Device linkage There is no cryptographic link between identities and device keys. In other words, Eve can attach her own device identity key as if it is a resource belonging to Bob and fool Alice into adding it.

[8]Which also means that the payload ciphertext must be known when the header is sent, which is problematic for on-the-fly encryption. See also Section 2.3.

There is a solution: each device could sign a certificate on each device identity key of the same user. While Eve might fool Alice into thinking that Bob has another device, it is highly unlikely that Bob is tricked into accepting another device as his own. Device identity keys with a certificates that was signed by an already accepted device of the same user could be accepted automatically.

In order to account for compromised devices, users must have the ability to revoke certificates and certificates should have a finite lifetime. This solution can be extended into a full-blown public key infrastructure (PKI) or web of trust, but I recommend to keep that out of the scope of the OMEMO specification (although compatibility with such systems could be taken into account when updating the OMEMO specification).

2.2.4 Forward/future secrecy

The forward secrecy and future secrecy of the protocol might be affected in unexpected ways when a user has read-only devices or inactive devices.

Read-only devices Read-only devices will forward their Signal chaining key, but never is there any message sent from these devices, so the Signal root key will never be ratcheted forward. Such a device compromises the future secrecy of the entire conversation: if the receiving chaining key of such a device gets compromised, the rest of the conversation from that point on is compromised.

The solution is simple, the read-only device should regularly send a KeyTransportElement in order to forward the ratchet. The interval for this message can be based on a number of received messages, on time, or on a combination of these.

Inactive devices Devices that are no longer used and never come online anymore, should be pruned from the conversation: they keep a copy of a very old chain key in their memory, which compromises the forward secrecy of the entire conversation. There is currently no way specified for removing keys from a conversation, except for just removing them.

A device can interpret the above message for read-only devices as an authenticated heartbeat message. When the device has not not received a heartbeat for too long, it can decide to prune the device from the conversation.

2.3 OMEMO Encrypted Jingle File Transfer

The OMEMO Encrypted Jingle File Transfer is defined in its ProtoXEP [11]. It uses the Jingle File transfer [25] to send the data to the other user. The KeyTransportElement is included in the Jingle File description and the file contents can be sent separately, encrypted with the random key that was sent in the KeyTransportElement.

From a cryptographic perspective, there is no difference between sending an OMEMO text message and sending an OMEMO-encrypted Jingle file, even if that file gets sent over another channel. The one difference is that Jingle allows for some file metadata to be sent. This metadata is neither encrypted nor authenticated. The specification does not provide a method for encrypting the metadata as well.

Message authentication Just as a normal message is not authenticated in the presence of a malicious device (see Section 2.2.3), so is the file content not authenticated when a malicious device is present.

The earlier proposed solution for authenticating the payload (authenticating the ciphertext in the SignalMessage tag) would disable on-the-fly encryption when sending a file, because the payload ciphertext must be known when constructing the `<header/>`. If losing the ability to do on-the-fly encryption is acceptable, this solution should be preferred. Otherwise, just authenticating the list of all recipient devices should be sufficient to protect against the described attack.

Metadata Even though the metadata is not secured by the specification, it should not leak any information on the raw file contents. The Jingle protocol requires a hash of the file. The OMEMO file-transfer specification is correct in requiring that this hash is of the file ciphertext: a plaintext hash would lead to a "confirmation-of-data" vulnerability [31].

All other metadata can simply be removed from the `<description/>` in order to minimize metadata leakage, as they are considered optional for Jingle. However, the "size" and "range" elements can be included, as these already leak from the ciphertext length and the transfer method.

3 Code Review

Conversations is an open-source [8] XMPP client for Android. In this section, I will use their published code as a reference implementation for the OMEMO ProtoXEP. I have inspected the implementation, looking for bugs that compromise the security of an OMEMO session in any way. The goal of the code review is twofold: it attempts to find security weaknesses and it should reveal if inconsistencies exist between the specification and its implementation. In the rest of this session I will give a summary of my findings.

The Conversations code simply uses the Signal library by OWS. Generation of Signal keys, encryption of `<key/>` elements and managing of the corresponding Signal sessions is handled by the Signal library. The biggest problem with this approach was that the Signal library accepted messages without a one-time prekey, which OMEMO should never do (since the server will never "run out" of one-time prekeys).[9] Combined with the fact that the signed prekeys never get removed/updated, this means that there was no *forward secrecy* for PreKeySignalMessages.

Key generation for the Signal keys (identity key, prekeys and ephemeral keys) is handled by the Signal library. The random key for the OMEMO payload is generated by `javax.crypto.KeyGenerator` class, instantiated for 128 bits AES and a 128 bit payload IV is generated by `java.security.SecureRandom`.

The Conversations application does not keep prekeys in memory during a MAM catch-up. Instead, the application uses the Signal library, which always deletes the keys from the store after decryption of a PreKeySignalMessage.

HTTP file upload Instead of using the OMEMO encrypted Jingle File Transfer as a default method for file transfer, the application gives preference to HTTP upload [12]. That setup adds another layer of indirection: the file is encrypted using AES in GCM mode, using a random 128 bit key and a 64 bit IV, both generated by the `java.security.SecureRandom` class. The file is then uploaded and the sender gets an URL. The used key and IV are appended to the URL as fragment identifier. The full URL is then considered to be the payload of the OMEMO MessageElement. This is not necessarily wrong (a HTTP client should not send the

[9] The developers fixed this in commit cc209af.

fragment identifier to the server in the HTTP request), but it is not a clean solution and there is a significant chance that some other client will get this wrong. In addition, the additional layer of indirection suffers from the same problem when a malicious device is present: it offers no authentication of the file content. To fix this, both the OMEMO payload and the file would have to be authenticated inside the Signal session.

X509 certificates The code allows X509 certificates on identity keys, although this is currently disabled by default. I have not looked in to much detail, as this is outside the scope of the OMEMO specification, but there appears to be nothing wrong with this approach.

Purge The conversations application allows users to purge the key of other devices, which says that it irreversibly marks the key as compromised. This irreversibility is not guaranteed and is only enforced by the fact that the application provides no user interface for reversing. Users have no method for purging their own keys or otherwise marking them as compromised.

Group messages The Conversations application allows for group conversations, although this is not specified by the ProtoXEP. From a cryptographic perspective, these multi-user chats are no different from a multi-device chat: to send a message to all users, the sending device will have to set up a Signal session with each of the participating devices, regardless of the user to which the device belongs.

4 Conclusions/recommendations

The OMEMO standard provides a protocol for secure communication with multiple devices. This protocol is only secure if both users apply good operational security in securing their devices and in adding devices of the other party.

When both users are careful, they can set up a secure multi-device session. However, if one of the users makes a mistake and adds a malicious device, or if just one device of the users gets compromised, the authentication of all messages is compromised, which is not necessary. The (ciphertext of the) payload should be authenticated in each SignalMessage, preferably as AAD.

The current OMEMO specification provides no link between devices that belong to the same user. Eve might trick Alice thinking that her key belongs to Bob. Bob should be able to sign a certificate that tells Alice which devices belong to him, she would not be tricked so easily by Eve.

Each devices should regularly send a message (a heartbeat) in order to forward the root ratchet of the Signal sessions, so that future secrecy can be ensured. The already existing KeyTransportElement can be used as an empty message that achieves this functionality.

Inactive devices, devices that never come online anymore, should be removed from a conversation by the owning user. Their presence in a conversation means that the forward secrecy of the entire conversation is compromised, because they hold on to an old key. In addition, I recommend that inactive devices may be removed by the other user. The above described heartbeat would provide users with a method for detecting if a device has become inactive.

The lifetime of (signed) prekeys should be mentioned in the standard. Signed prekeys should be changed regularly in order to achieve forward secrecy. This should at least be done after every time the user receives a PreKeySignalMessage that uses the latest signed prekey, but it can be done more often (based on time) to ensure the forward secrecy of dropped messages. The standard should allow for alternative ciphers. However,

the standard should limit itself to the ciphers used in the OMEMO encryption. Signal also has no way for specifying ciphers, but it is not in the scope of the OMEMO standard to specify that.

Prekey collisions can be greatly reduced if the server hands out each key only once, instead of all keys to every user that asks. This would not affect security, but it would make successful delivery of the first message of the protocol more reliable.

The specification should update its terminology to reflect the recent name changes by Open Whisper Systems. Specifically, the term “Axolotl” should be replaced with “the Signal Protocol” and the message names “PreKeyWhisperMessage” and “WhisperMessage” should be replaced with “PreKeySignalMessage” and “SignalMessage”.

My final remark is about the reference implementation. Unless a change is made in the way that servers provide the keys, the code should not accept PreKeySignalMessages without a one-time prekey. As stated before, this has already been fixed in commit cc209af.

5 Acknowledgement

I would like to thank Daniel Gultsch for helping me out with some of the questions I have had on the protocol and for his quick processing of my feedback in the Conversations code.

6 References

[1] Daniel J. Bernstein. *Public Key Cryptography - PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24- 26, 2006. Proceedings*, chapter Curve25519: New Diffie-Hellman Speed Records, pages 207–228. Springer Berlin Heidelberg, Berlin, Heidelberg , 2006. https://cr.yp.to/papers.html#curve25519.

[2] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang. High-speed high-security signatures, *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. https://ed25519.cr.yp.to/.

[3] Daniel J. Bernstein, Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. http://safecurves.cr.yp.to. Accessed: 2015-05-04.

[4] Jean Paul Degabriele, Anja Lehmann, Kenneth G. Paterson, Nigel P. Smart, Mario Strefler. On the joint security of encryption and signature in emv, Cryptology ePrint Archive. Report 2011/615, 2011. https://eprint.iacr.org/2011/615.

[5] Danny Dolev, Andrew C. Yao. On the security of public key protocols, *Information Theory, IEEE Transactions on*, 29(2):198–208, March 1983.

[6] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jrg Schwenk, Thorsten Holz. How Secure is TextSecure?, Cryptology ePrint Archive. Report 2014/904, November 2014. http://eprint.iacr.org/2014/904.

[7] Daniel Gultsch. Conversations. https://github.com/siacs/Conversations. Accessed: 2016-04-07.

[8] Daniel Gultsch. Conversations is an open source XMPP/Jabber client for Android 4.0+ smart phones. https://github.com/siacs/Conversations. Accessed: 2016-05-10.

[9] Daniel Gultsch. Conversations: the very last word in instant messaging. https://conversations.im/. Accessed: 2016-04-07.

[10] Daniel Gultsch. OMEMO Multi-End Message and Object Encryption. https://conversations.im/omemo/. Accessed: 2016-04-07.

[11] Daniel Gultsch. XEP-xxxx: OMEMO Encrypted Jingle File Transfer. ProtoXEP, XMPP Standards Foundation, September 2015. https://xmpp.org/extensions/inbox/omemo-filetransfer.html.

[12] Daniel Gultsch. XEP-0363: HTTP File Upload. Standards Track, XMPP Standards Foundation, March 2016. https://xmpp.org/extensions/xep-0263.html.

[13] Joe Hildebrand, Matthew Miller. XEP-0280: Message Carbons. Standards Track, XMPP Standards Foundation, February 2016. https://xmpp.org/extensions/xep-0280.html.

[14] Hugo Krawczyk, Mihir Bellare, Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, RFC Editor, February 1997. https://www.rfc-editor.org/rfc/rfc2104.txt.

[15] Hugo Krawczyk, Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, RFC Editor, May 2010. https://www.rfc-editor.org/rfc/rfc5869.txt.

[16] Moxie Marlinspike. Advanced cryptographic ratcheting. November 2013. https://whispersystems.org/ blog/advanced-ratcheting/. Accessed: 2016-05-10.

[17] Moxie Marlinspike. Forward Secrecy for Asynchronous Messages. Augustus 2013. https://whispersystems.org/blog/asynchronous-security/. Accessed: 2016-05-10.

[18] Moxie Marlinspike. Simplifying OTR deniability. July 2013. https://whispersystems.org/blog/ simplifying-otr-deniability/. Accessed: 2016-05-10.

[19] Moxie Marlinspike. Private Group Messaging. May 2014. https://whispersystems.org/blog/private-groups/. Accessed: 2016-04-07.

[20] Moxie Marlinspike. Signal on the outside, Signal on the inside. March 2016. https://whispersystems.org/blog/signal-inside-and-out/. Accessed: 2016-04-07.

[21] Kevin Smith, Matthew Wild. XEP-0313: Message Archive Management. Standards Track, XMPP Standards Foundation, March 2016. https://xmpp.org/extensions/ xep-0313.html.

[22] Alfred Menezes, Berkant Ustaoglu. On reusing ephemeral keys in Diffie-Hellman key agreement protocols, *International Journal of Applied Cryptography*, 2(2):154–158, 2010.

[23] NIST. Announcing the Advanced Encryption Standard (AES). *Technical report, NIST*, November 2001.

[24] Trevor Perrin. Double Ratchet Algorithm. https://github.com/trevp/doubleratchet/wiki. Accessed: 2016-04-07.

[25] Lance Stout, Peter Saint-Andre. XEP-0234: Jingle File Transfer. *Standards Track, XMPP Standards Foundation*, March 2016. https://xmpp.org/extensions/xep-0234.html.

[26] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. *RFC 6120, RFC Editor*, March 2011. https://www.rfc-editor.org/rfc/rfc6120.txt.

[27] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. *RFC 6122, RFC Editor*, March 2011. https://www.rfc-editor.org/rfc/rfc6122.txt.

[28] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Mes- saging and Presence. *RFC 6121, RFC Editor*, March 2011. https://www.rfc-editor.org/ rfc/rfc6121.txt.

[29] Andreas Straub. XEP-xxxx: OMEMO Encryption. *ProtoXEP, XMPP Standards Foundation*, October 2015. https://xmpp.org/extensions/inbox/omemo.html.

[30] Open Whisper Systems. Signal Protocol library for Java/Android. https://github.com/ WhisperSystems/ libsignal-protocol-java. Accessed: 2016-05-10.

[31] Zooko Wilcox-O'Hearn. Attacks on Convergent Encryption. *Technical report, Tahoe-LAFS*, March 2008. https://tahoe-lafs.org/hacktahoelafs/drew perttula.html. Accessed: 2016-05-10.

Appendix 1 Minor corrections

During my review of the OMEMO documentation, I noted some minor errors in the specification, most of which are typographical errors. This appendix contains a list of corrections. None of these errors affect the security of the protocol in any way.

In the OMEMO XEP:

- Section 4.5: both own devices (should be: both owned devices)
- Section 6: axoltol (should have been: axolotl; should be: “the Signal Protocol”)
- Appendix G: duplicate references
- Inconsistent usage of “.” (period) at the end of list items

In the OMEMO file transfer XEP:

- Section 3: Remeo and Juliet (should be: Romeo and Juliet)
- Section 3: file tranfer (should be: file transfer)
- Section 3, Example 1: `</file>` has wrong indentation
- Section 5: intilization (should be: initialization)
- Section 5: the hash of encrypted file (should be: the hash of the encrypted file)
- Section 5: rangend tranfer (should be: ranged transfer)
- Section 7: might not the Device ID (should be (?): might not have)
- Section 8: Last list item is missing a “.” (period)
- The document is missing a reference to the OMEMO XEP