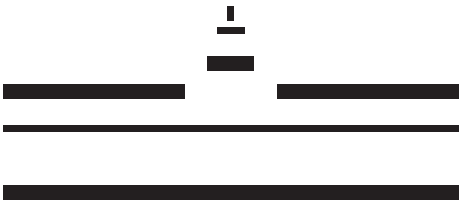

Westfälische
Wilhelms-Universität
Münster

Untersuchung von Chat-Verschlüsselung am Beispiel einer OMEMO-Implementation für Smack

Bachelorarbeit

eingereicht von:

Paul Moritz Schaub

Matrikelnummer: 406218

Abschluss: B. Sc. Informatik

Betreut von:

Dr. Dietmar Lammers

Münster, 23. Februar 2017

1 Aufgabenstellung

1.1 Praxis

Ich werde im Rahmen meiner Bachelorarbeit das OMEMO-Verschlüsselungsprotokoll mit Hilfe der *Signal*-Bibliothek[1] für die XMPP-Bibliothek *Smack*[2] implementieren. Alle frühen Implementationen von OMEMO verwenden diese für Verschlüsselungsroutinen (siehe Conversations[3], gajim-omemo[4]). Im Dezember 2016 wurde OMEMO jedoch als XEP-0384 mit Olm[5], statt Signal als Unterbau spezifiziert[6] (Anmerkung: Das hat sich inzwischen erneut geändert). Da es jedoch zum jetzigen Zeitpunkt noch keine Java Bibliothek gibt, die Olm implementiert, verwende ich weiterhin die Signal Bibliothek libsignal-protocol-java[1] von *Open Whisper Systems*. In meiner Implementation halte ich die Verschlüsselungsbibliothek jedoch in einem zweiten Modul gekapselt, um einen späteren Wechsel auf Olm oder eine andere Bibliothek zu erleichtern. Dieser Schritt wird später auch notwendig sein, um das Modul in Smack integrieren zu können, da Smack unter der APL lizenziert ist[7], libsignal-protocol-java jedoch unter der GPL steht[8], was zu einer Lizenzunverträglichkeit führt.

1.2 Theorie

Im theoretischen Teil meiner Arbeit werde ich das OMEMO- bzw. Signal-Protokoll untersuchen. Ich möchte zunächst einen kurzen Einstieg in die Kryptografie geben und Grundprinzipien wie *Public-Key-Kryptografie*, *Schlüsselaustausch-* und *Signaturmechanismen* geben. Im zweiten Teil werde ich die Funktionsweise und Eigenschaften von OMEMO, bzw. dem zugrundeliegenden Signal Protokoll erläutern und Vergleiche zu herkömmlichen Protokollen wie OpenPGP und OTR[9] im Kontext der Nutzung mit mehreren (synchronisierten) Geräten pro Nutzer anstellen. Dort werde ich erläutern, für welche Einsatzgebiete die herkömmlichen Verfahren entwickelt wurden, wie sich das Kommunikationsökosystem seither gewandelt hat und welche Probleme sich durch diesen Wandel ergeben. Es sollte klar werden, wie OMEMO diese Probleme im Kontext von Instant Messaging über das XMPP Netzwerk löst. Außerdem werde ich kurz erläutern, welche Schritte ein Angreifer unternehmen könnte und welche Gegenmaßnahmen das OMEMO Protokoll verwendet.

Plagiatserklärung der / des Studierenden

Hiermit versichere ich, dass die vorliegende Arbeit über _____ _____ selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommenen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

_____
(Datum, Unterschrift)

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

_____
(Datum, Unterschrift)

1.3 Danksagung

Als erstes möchte ich meinem Betreuer Dr. Dietmar Lammers und der Universität Münster dafür danken, dass sie mir den Ausflug zum *XSF*-Summit ermöglicht haben.

Ich danke Florian Schmaus für seine Ratschläge bezüglich der Smack Bibliothek und für die nette Einladung zum *XSF*-Summit.

Außerdem danke ich Daniel Gultsch, Andreas Straub und Philipp Hörist für ihre Unterstützung und Geduld bei Fragen bezüglich des OMEMO Protokolls, sowie Germán Márquez Mejía, René Calles und Richard Bayerle für Anregungen und Diskussionen.

Des weiteren danke ich der *XSF* und ihren Mitgliedern dafür, dass sie mich am 21. Summit in Brüssel teilnehmen ließen, und mir so einen tieferen Blick in die Arbeit der *XMPP Standards Foundation* ermöglicht haben.

Zuletzt möchte ich mich bei Kim für ihre Geduld bedanken.

"What if I got hit by lightning
while walking with an umbrella?
Ban umbrellas!
Fight the menace of lightning!"

CORY DOCTOROW, LITTLE BROTHER

"When crypto is outlawed,
only outlaws will have crypto."

PHIL ZIMMERMAN

Cryptography is the ultimate form
of non-violent direct action."

JULIAN ASSANGE

Inhaltsverzeichnis

2 Einleitung

2.1 Kurzübersicht Kryptografie

Kryptografie ist heutzutage allgegenwärtig. Häufig bekommen wir es gar nicht mit, dass im Hintergrund unserer Handlungen kryptografische Prozesse stattfinden. Vor allem, wenn wir das Internet nutzen, haben wir es direkt oder indirekt mit Verschlüsselung zu tun. Ein prominentes Beispiel ist Online-Banking, jedoch kommt Kryptografie schon bei grundsätzlichen Aktivitäten, wie dem einfachen Nachrichtenaustausch zum Einsatz. Ich werde mich im Rahmen meiner Bachelorarbeit auf das Szenario der Mensch-zu-Mensch Kommunikation beschränken. Konkret soll es um Online-Chats über das XMPP-Protokoll gehen. Doch zunächst müssen jedoch ein paar grundsätzliche Begrifflichkeiten geklärt werden.

2.1.1 Was ist *Kommunikation*?

Unter Kommunikation verstehen wir den Informationsaustausch zwischen zwei oder mehr Kommunikationspartnern. Zur Veranschaulichung können wir das Kommunikationsmodell von Shannon und Weaver heranziehen (Abbildung 2.1), jedoch können wir die Störungsquelle vernachlässigen, da sie für unseren Fall nicht von Bedeutung ist.

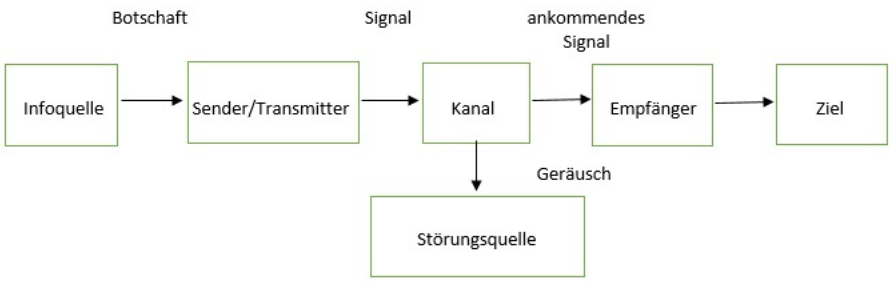

Abbildung 2.1: Kommunikation nach Shannon & Weaver.
Von *Anotherbadcode* - Eigenes Werk, CC-BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=46700510

Bei den ausgetauschten Informationen kann es sich z.B. um Textnachrichten handeln, jedoch kann der Nachrichtenaustausch auch in Form von z.B. Telefonaten oder Videokonferenzen stattfinden. Der Begriff *Information* ist im Kontext der Kommunikation sehr weitreichend. Nachrichtenaustausch geschieht im allgemeinen sequenziell und über einen Informationskanal. Informationen werden in *Nachrichten* gekapselt. Eine Nachricht kann z.B. ein Brief sein, eine Chatnachricht oder ein Datenpaket. Der Informationskanal kann dementsprechend z.B. der Postweg oder das Internet sein.

2.1.2 Was sind die Ziele der Kryptografie?

In der Literatur bezeichnet man die Kommunikationspartner oft mit den Pseudonymen *Alice* und *Bob*[10]. Eine lauschende dritte Partei wird stellvertretend mit dem Namen *Eve* oder *Mallory* bezeichnet. Ich verwende in meiner Arbeit Ersteres. In der Kryptografie geht es vorrangig darum, zu verhindern, dass *Eve* Zugriff auf den Inhalt der zwischen *Alice* und *Bob* ausgetauschten Nachrichten bekommt. Es wird davon ausgegangen, dass *Eve* vollen Zugriff auf das zur Kommunikation verwendete Medium, also den Kommunikationskanal hat. *Eve* ist damit in der Lage alle gesendeten Nachrichten abzufangen, mitzulesen und verändert weiterzugeben (vgl. Dolev-Yao-Modell[11]). In der Kryptografie werden dementsprechend eine Reihe von Zielen definiert, die mithilfe von kryptografischen Verfahren erreicht werden sollen. Die drei wichtigsten Ziele umfassen *Vertraulichkeit*, *Integrität* und *Authentizität*[12].

Unter *Vertraulichkeit* versteht man, dass nur *Alice* und *Bob* den Inhalt der Nachricht lesen können. Dazu wird die ausgetauschte Information mithilfe eines Schlüssels verschlüsselt. Eine verschlüsselte Nachricht kann nur von Personen mit Kenntnis des passenden geheimen Schlüssels entziffert werden. Darauf werde ich im weiteren Verlauf der Arbeit genauer eingehen.

Die *Integrität* einer Nachricht ist dadurch definiert, dass *Bob* sich sicher sein kann, dass eine Nachricht, die er von *Alice* erhält, in genau der selben Form von Alice verschickt wurde und auf dem Weg nicht durch Übertragungsfehler oder gar böswillig durch *Eve* verändert wurde. Erreicht wird dieses Ziel meist durch die Nutzung von Prüfsummen (*MD*, engl. *Message Digest*).

Authentizität bedeutet, dass *Bob* sich sicher sein kann, dass eine Nachricht von *Alice* auch *wirklich* von *Alice* stammt. Damit soll verhindert werden, dass *Eve* Nachrichten im Namen eines der Kommunikationspartner fälscht, oder sich als diesen ausgibt. Um Authentizität zu gewährleisten, nutzen viele Systeme kryptografische Signaturen.

Des Weiteren werden oft folgende zusätzliche Ziele definiert:

Glaubhafte Abstreitbarkeit (auch *plausible deniability*) bedeutet, dass *Bob* *Alice* zu keinem Zeitpunkt nach der Kommunikation, gegenüber einer dritten

Partei nachweisen kann, dass eine empfangene Nachricht wirklich von *Alice* verfasst wurde[13, S. 2]. *Alice* kann also *abstreiten* Autor einer Nachricht zu sein. Dieses Ziel scheint im ersten Moment im Widerspruch zum Prinzip der *Authentizität* zu stehen, da *Bob* sich dadurch ja sicher sein kann, dass er mit *Alice* kommuniziert, doch Protokolle wie OTR (vgl. Abschnitt 3.5.2) und auch OMEMO (Abschnitt 3) erfüllen beide Ziele gleichzeitig. *Plausible deniability* wird u.a. dadurch erreicht, dass nach der Kommunikation die zur Nachrichtensignatur genutzten Schlüssel veröffentlicht werden, sodass beide Parteien theoretisch in der Lage sind, den Nachrichtenverlauf zu fälschen[14].

Durch *Perfect Forward Secrecy* (etwa *vorwärtsgerichtete Geheimhaltung*, kurz *PFS*) möchte man verhindern, dass *Eve* im Falle eines erfolgreichen Angriffs auf *Alice* und *Bob*'s Kommunikation Zugriff auf die Inhalte bereits gesendeter Nachrichten (z.B. vergangene Gespräche) bekommt[15, S. 184]. Durch *PFS* wird also sichergestellt, dass alle Kommunikation *vor* dem Zeitpunkt des erfolgreichen Angriffes vor *Eve*'s Augen sicher ist. PFS wird häufig dadurch erzielt, dass Schlüsselmaterial, mit dem vergangene Kommunikation entschlüsselt werden kann, verworfen wird und in regelmäßigen Abständen neue Schlüssel generiert werden.

Future Secrecy (etwa Zukunftssicherheit, auch *break-in recovery*) hingegen bezeichnet das Ziel, Kommunikation, die *nach* einem erfolgreichen Angriff stattfindet, vor dem Angreifer zu sichern. Man möchte somit verhindern, dass ein Angreifer seinen Lauschangriff beliebig lange fortführen kann. Der Algorithmus *erholt* sich sozusagen von dem Angriff. Auch diese Eigenschaft wird häufig durch die Einführung neuer Schlüssel erzielt. Insbesondere wird zum erstellen eines neuen Schlüssel die Ausgabe eines Zufallsgenerators mit einbezogen, welche der Angreifer nicht kennt. Mehr dazu folgt in Abschnitt 3.2.2.

Eine sehr wünschenswerte Form der Verschlüsselung ist sogenannte *Ende-zu-Ende-Verschlüsselung* (E2E). Eine Nachricht durchquert in den meisten Fällen einige Stationen bevor sie beim gewünschten Empfänger ankommt. Zu diesen Stationen zählt z.B. der genutzte Server. Im Gegensatz zur *Punkt-zu-Punkt-Verschlüsselung* (P2P), bei der die Nachricht von jedem Punkt in der Kette nur für den nächsten Punkt verschlüsselt, dort wieder ent- und wieder verschlüsselt wird, wird die Nachricht bei Ende-zu-Ende-Verschlüsselung schon beim Sender für den Empfänger verschlüsselt, sodass die Nachricht an Zwischenstationen nicht gelesen werden kann. Im Gegensatz zur P2P-Verschlüsselung braucht man dem Server nicht zu vertrauen.

2.1.3 Public-Key-Kryptografie

Betrachten wir folgendes Szenario: *Alice* möchte eine Nachricht an *Bob* senden und gleichzeitig verhindern, dass *Eve* mitlesen kann. Sie möchte die Nachricht also verschlüsseln. Dazu benutzt sie einen Schlüssel, der geheimgehalten wird

und mit dessen Hilfe sie oder *Bob* die Nachricht wieder entschlüsseln können (vgl. Abbildung 2.2).

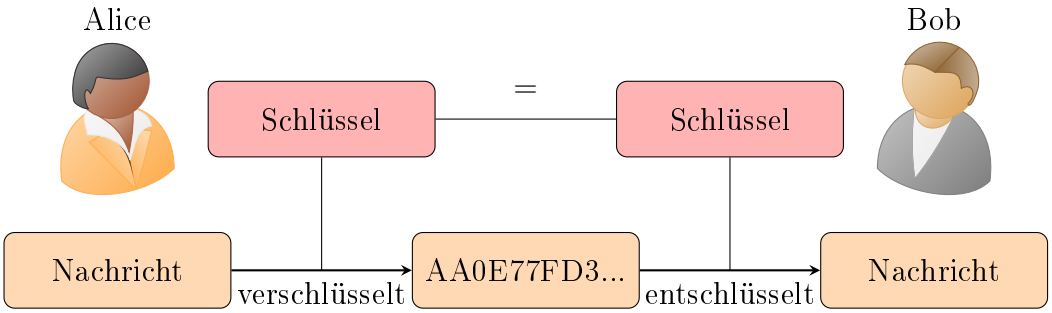

Abbildung 2.2: Symmetrische Verschlüsselung

Wenn *Alice* und *Bob* die Möglichkeit haben sich privat zu treffen, um den Schlüssel (z.B. ein Passwort) zu vereinbaren, ist das auch kein Problem. Nur was ist, wenn die beiden weit voneinander entfernt leben und sich nicht im Privaten treffen können? Wie übergibt *Alice* den Schlüssel?

Stellen wir uns das oben genannte Szenario noch einmal leicht verändert vor. *Alice* hat einen Brief (die geheime Nachricht) und eine Kiste mit Vorhängeschloss samt Schlüssel (das Verschlüsselungsverfahren). *Alice* kann den Brief in die Kiste legen und das Schloss mit ihrem Schlüssel verschließen. Nur wie bekommt *Bob* diesen nun, um die Kiste wieder zu öffnen? Um dieses Problem zu lösen, können beide wie folgt handeln: *Bob* schickt *Alice* sein Schloss (geöffnet). *Alice* kann nun den Brief in die Kiste legen und diese mit dem Schloss verschließen, das sie von *Bob* erhalten hat. Sie schickt die Kiste zu *Bob* zurück und dieser kann die Kiste mit seinem Schlüssel öffnen. Damit wurde das Problem der Übertragung des Schlüssels umgangen.

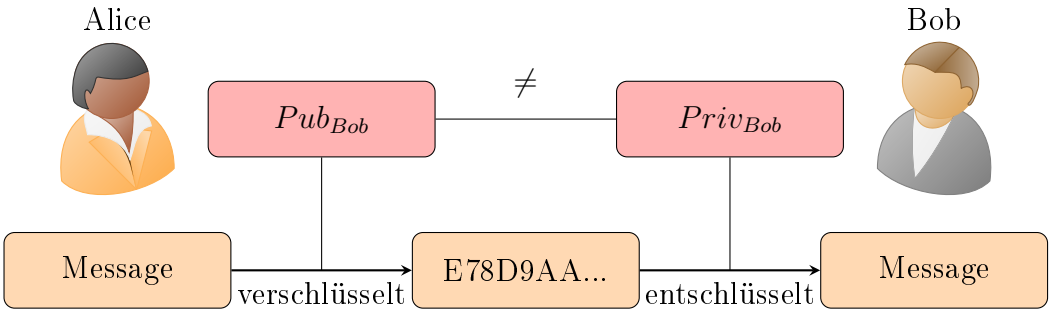

Abbildung 2.3: Asymmetrische Verschlüsselung mit Public Key Kryptografie

Dieses Prinzip wird *Public-Key-Kryptografie*[16] genannt. Statt das selbe Geheimnis (z.B. Passwort) sowohl für die Ver- als auch Entschlüsselung zu nutzen (in dem Fall spricht man von symmetrischer Verschlüsselung[17]), nutzt man ein Schlüsselpaar (symbolisch Schlüssel und Schloss). Der geheime Schlüssel

(*private key*) muss vom Eigentümer geheimgehalten werden. Der sogenannte *öffentliche* Schlüssel (*public key*) hingegen wird frei verteilt. Wenn nun *Alice* eine Nachricht an *Bob* senden möchte, besorgt sie sich zunächst *Bob*'s *öffentlichen* Schlüssel. Diesen nutzt sie, um ihre Nachricht zu verschlüsseln und sendet sie dann an *Bob*. Der nutzt dann seinen *geheimen* Schlüssel, um an den Inhalt der Nachricht zu gelangen (vgl. Abbildung 2.3). Ein Beispiel für *Public-Key-Kryptografie* ist das sogenannte *RSA*-Verfahren[16]. Bemerkenswert ist, dass die Schlüssel eines Schlüsselpaares eindeutig zueinander gehören[16]. Jeder öffentliche Schlüssel hat genau ein privates Gegenstück und umgekehrt. Jedoch kann man nicht aus einem der beiden Rückschlüsse auf den anderen ziehen.

2.2 XMPP

2.2.1 Extensible Messaging and Presence Protocol

In meiner Bachelorarbeit befasse ich mich mit dem Protokollstandard XMPP (auch *Jabber*, von der *IETF* spezifiziert als RFC 6120[18], 6121[19], 6122[20]). XMPP (kurz für *Extensible Messaging and Presence Protocol*) wurde vor allem für den Austausch von Textnachrichten und Statusmeldungen entwickelt[21], jedoch sind auch viele andere Nutzungsfälle möglich (z.B. IoT). Das Protokoll ist inzwischen mehr als 15 Jahre alt. Es gibt ein ganzes Netzwerk von dezentralisierten Servern, die über den gesamten Erdball verteilt sind[21]. *Dezentralisiert* bedeutet, dass es keinen Zentralen Server gibt, der Nachrichten verteilt, sondern dass alle am XMPP Netzwerk teilnehmenden Server gleichberechtigt sind. XMPP ermöglicht durch *Federation*, dass auch Nutzer unterschiedlicher Server miteinander kommunizieren können (ähnlich wie bei E-Mail)[22, S. 12]. Zusätzlich können externe Dienste wie ICQ, Matrix oder IRC mit sogenannten *bridges* ins XMPP Netzwerk eingebunden werden. Verwaltet wird die Spezifikation von der Internet Engineering Taskforce (IETF) und der XMPP Standards Foundation (XSF)[23]. Durch sogenannte XEPs (XMPP Extension Protocols) kann das Protokoll um Funktionen ergänzt werden. So ist zum Beispiel der Empfang von Nachrichten, die versandt wurden, während der Empfänger offline war, durch ein XEP spezifiziert.

Im XMPP-Protokoll besitzt ein Nutzer eine sogenannte *Jabber-ID* (kurz *JID*), die einer E-Mailadresse gleicht[22, S. 14]. *Alice* könnte beispielsweise die JID *alice@wonderland.lit* besitzen. Unterschieden wird zwischen Benutzernamen (*alice*) und dem Server, auf dem *Alice* sich registriert hat (*wonderland.lit*). Es ist möglich, dass *Alice* sich gleichzeitig mit mehreren Geräten anmeldet. Um diese zu unterscheiden, wird für jedes Gerät ein eigener Postfix angehängt (z.B. *alice@wonderland.lit/laptop* für *Alice*' Laptop oder *alice@wonderland.lit/phone* für ihr Mobiltelefon). Diesen sogenannten *Ressourcenbezeichner* kann Alice pro Gerät frei festlegen.

Wenn nun *Bob* mit seinem Account *bob@marley.jm* eine Nachricht an *Alice* schickt und beide Server miteinander federieren, geht die Nachricht zunächst von *Bob*'s Gerät an seinen Server *marley.jm*, von dort aus zu *wonderland.lit* und schließlich zu *Alice* (siehe Abbildung 2.4).

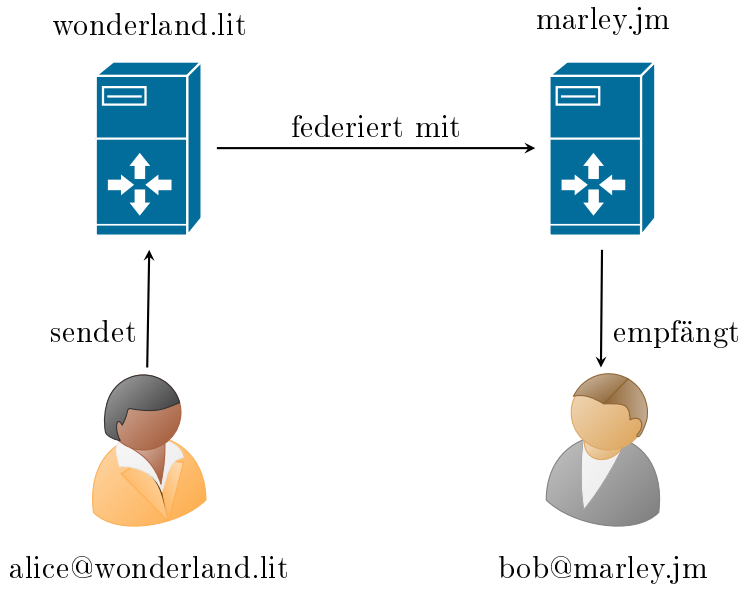

Abbildung 2.4: Funktionsweise federierter Nachrichtenübertragung

Neben Gesprächen unter vier Augen (*peer-to-peer*) unterstützen viele Clients Konferenzen mit mehreren Nutzern, sogenannte MUCs (*Multi-User-Chats*). MUCs sind spezifiziert als XEP-0045[24].

XMPP Pakete (auch *Stanzas* genannt) sind XML Bausteine, welche zwischen Server und Client, bzw. Server und Server ausgetauscht werden[22, S. 16]. Durch XEPs können beliebige Datenstrukturen definiert werden, was die hohe Flexibilität von XMPP möglich macht[21]. Einige Beispiele bezüglich OMEMO finden sich in Abschnitt 3.4.3.

2.2.2 Smack - Java Bibliothek für XMPP

Smack[2] ist eine freie, modulare Java Bibliothek, die viele Funktionen zur Nutzung des XMPP Netzwerkes zur Verfügung stellt. Die Funktionalitäten der Bibliothek sind aufgeteilt in viele einzelne Module, was ein hohes Maß an Flexibilität ermöglicht. Smack ist außerdem sowohl auf Systemen mit Java SE 7, als auch unter Android lauffähig[25]. Auf Smack basieren zum Beispiel die Messenger *Xabber*, *Yaxim*, *ChatSecure* (Android), *Kontalk*, *anderChat*, *Jitsi*, *Zom* und *Spark*[26, 27, 28, 29, 30, 31, 32, 33]. Die Bibliothek ist unter der Apache Public License lizenziert[7]. Meine Implementation basiert auf der aktuellen Beta Version 4.3.

3 OMEMO

OMEMO ist ein rekursives Akronym für *"OMEMO Multi-End Message and Object Encryption"*. Das Protokoll wurde 2015 von Andreas Straub im Zuge eines *Google Summer of Code* (GSoC) mit dem Ziel entwickelt, das Signalprotokoll (ehemals *Axolotl*[34]) für den XMPP basierten freien Android Messenger *Conversations*[3] zu implementieren[35].

Das Signalprotokoll wurde 2013 von Trevor Perrin und Moxie Marlinspike entwickelt und in *OpenWhisperSystems* freiem Android Messenger *TextSecure* (später *Signal*)[34] eingeführt. Die kryptografischen Eigenschaften und Bestandteile des Signaprotokolls (namentlich der *Double Ratchet Algorithmus*, das *X3DH Schlüsselaustauschprotokoll* sowie die *XEdDSA* und *VXEdDSA Signaturverfahren*) werden weiter unten genauer beleuchtet.

OMEMO ist eine Art *Wrapperprotokoll*, welches das Signalprotokoll für XMPP adaptiert[36, S. 6]. Signal verwendet kein XML zur Datenübertragung.

3.1 Extended Triple Diffie-Hellman Schlüsselaustausch Protokoll

Dieser Abschnitt schafft einen kleinen Überblick über das X3DH-Schlüsselaustauschverfahren, wie es von Marlinspike und Perrin in "The X3DH Key Agreement Protocol"[37] beschrieben wurde. Zunächst erläutere ich jedoch einige Grundfunktionen.

Der Diffie-Hellman-Schlüsselaustausch (auch Diffie-Hellman-Merkle Schlüsselaustausch, kurz *DH*) ist ein Verfahren, dass es zwei Kommunikationspartnern *Alice* und *Bob* ermöglicht, über einen unvertrauten Kanal einen geheimen Schlüssel zu vereinbaren, ohne dass ein eventueller Angreifer *Eve* Kenntnis von diesem Geheimnis erlangen kann. Dabei wird davon ausgegangen, dass *Eve* jede Nachricht, die zwischen *Alice* und *Bob* ausgetauscht wird, abhören und in der Form, wie sie übertragen wird, mitlesen kann[11].

3.1.1 Der klassische Diffie-Hellman-Merkle Schlüsselaustausch

Der klassische DH wurde 1976 von Whitfield Diffie und Martin Hellman unter der Bezeichnung *"ax1x2"* veröffentlicht[38]. Dabei handelte es sich um das erste *asymmetrische* Verschlüsselungsverfahren. Dem klassischen Verfahren zugrunde liegt das Problem des diskreten Logarithmus. In der Kryptografie werden häufig sogenannte "Einwegfunktionen"genutzt. Das bedeutet, dass die Funktion

$$f : X \to Y, f(x) = y$$

leicht in eine Richtung ausführbar ist, man jedoch *sehr* schwer von einem Ergebnis auf die Eingabe schließen kann:

$$f^{-1} : Y \to X, x = f^{-1}(y)$$

Veranschaulichen kann man sich das Prinzip einer Einwegfunktion mit dem Beispiel eines *sehr* großen Telefonbuches. Da die Einträge darin alphabetisch geordnet sind, ist das Finden einer Telefonnummer zu einem gegebenen Namen recht einfach. Zu einer Nummer jedoch den zugehörigen Namen zu finden, erweist sich als ziemlich aufwändig.

Die diskrete Exponentialfunktion

$$exp_{g,p}(x) = m, x \mapsto g^x \equiv m \mod p$$

ist eine solche Einwegfunktion[39]. Vom Eingabeparameter x und gegebenen festen g, p auf das Ergebnis m zu schließen ist einfach. Bei der Rückrichtung

$$exp^{-1}_{g,p}(m) = log_g(m) \text{ in } \mathbb{Z}_p$$

(dem diskrete Logarithmus) wird zu gegebenen g, p und m der eindeutige ganzzahlige Wert x gesucht, welcher die Gleichung der diskreten Exponentialfunktion erfüllt. Dies wird für riesige Werte schnell schier unmöglich. Da bisher kein effizienter Algorithmus zur Lösung dieses Problems bekannt ist, wird die diskrete Exponentialfunktion als *Einwegfunktion* bezeichnet.

Der klassische DH benutzt dieses Problem auf folgende Weise zur Lösung der Problematik des Schlüsselaustausches[38]:

Zunächst einigen sich *Alice* und *Bob* auf gemeinsame Parameter g und p, wobei p eine *sehr* große Primzahl ist und $g < p$ mit $g, p \in \mathbb{N}$. In der Praxis sind diese Werte meist fest vorgegeben, sodass *Alice* und *Bob* sich darüber nicht verständigen müssen[38]. Zusätzlich generieren beide jeweils eine geheime Zufallszahl. *Alice* nennt ihre a, *Bob* seine b. *Alice* berechnet nun $A = g^a \mod p$ und sendet A an *Bob*. Dieser berechnet wiederum $B = g^b \mod p$, sowie $G = A^b$

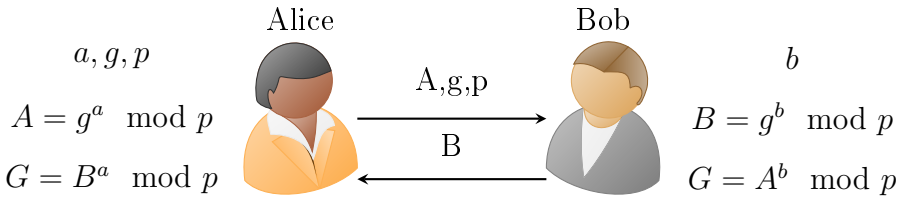

Abbildung 3.1: Diffie-Hellman-Merkle Schlüsselaustausch (eigener Entwurf nach Wikipedia)

mod p und schickt B zurück an *Alice*. Zuletzt berechnet *Alice* $G = B^a \mod p$ (vgl. Abbildung 3.1).

Beide besitzen nun das *gleiche* gemeinsame Geheimnis G und können dieses z.B. als Schlüssel für ein symmetrisches Verschlüsselungsverfahren nutzen. G ist auf beiden Seiten gleich, da

$$G_{Alice} = B^a = (g^b)^a = g^{(b \cdot a)} = g^{(a \cdot b)} = (g^a)^b = A^b = G_{Bob}$$

gilt. *Eve* kann nun jede gesendete Nachricht abfangen, jedoch keinerlei Rückschlüsse auf G ziehen, da G nie übertragen wurde und sie somit lediglich Kenntnis von g, p, A, B haben kann. Um auf a, b, und somit auf G schließen zu können, müsste sie den diskreten Logarithmus auf A oder B anwenden, um an a oder b zu gelangen. Dies ist jedoch, wie oben bereits erwähnt, *noch* nicht in realistischer Zeit möglich.

3.1.2 DH mit elliptischen Kurven

Um die Sicherheit des DH-Verfahrens zu steigern, kann der Algorithmus mit Hilfe von elliptischen Kurven (*elliptic curves*, kurz *EC*) implementiert werden. Ersetzt man den Modulus p durch eine elliptische Kurve (z.B. X25519 oder X448), und die Rechenoperationen Multiplikation und Potenzierung durch Punktaddition und Skalarmultiplikation auf elliptischen Kurven, so bleiben die Gleichungen intakt und das Verfahren funktioniert weiterhin[40]. Das n-fache Addieren eines Punktes zu sich selbst (nP) ersetzt somit die Potenzierung x^n im Diffie-Hellman-Merkle Schlüsselaustauschverfahren.

3.1.3 DH im Signalprotokoll

Das Signalprotokoll nutzt einen spezifischen Algorithmus zum Schlüsseltausch, das sogenannte *Extended Triple Diffie-Hellman*-Verfahren (kurz *X3DH*)[37]. Ein Problem des herkömmlichen DH ist, dass *Bob* aktiv reagieren muss, bevor *Alice* eine Unterhaltung mit ihm beginnen kann. Das erfordert unter anderem, dass *Bob* zum Zeitpunkt der Kontaktaufnahme zwingend online sein muss,

damit *Alice* den Schlüsselaustausch mit ihm durchführen kann, wenn sie ihm eine Nachricht senden möchte (vgl. *OTR*[9]). Diese Annahme ist im Kontext moderner Kommunikation nicht mehr zeitgemäß.

Das X3DH-Verfahren löst dieses Problem durch die Verwendung sogenannter *PreKeys*[37, S. 4]. Ein *PreKey* besteht aus einem EC-Schlüsselpaar über den Kurven X25519 oder X448. Jeder Kommunikationspartner veröffentlicht Schlüsselmaterial im Voraus, welches von anderen Nutzern jederzeit als erste Hälfte eines DH-Schlüsseltauschs genutzt werden kann, um eine Sitzung zu beginnen. Nehmen wir im folgenden an, *Alice* möchte eine Nachricht an *Bob* senden. In diesem Szenario veröffentlicht *Bob* zunächst sein Schlüsselmaterial. Es sei angemerkt, dass dieser Schritt an einem beliebigen Zeitpunkt geschieht *bevor Alice* ihm eine Nachricht senden möchte. Der Vollständigkeit halber und zum besseren Verständnis beschreibe ich diesen Schritt jedoch unmittelbar bevor *Alice* mit der Kommunikation beginnt. Ich berufe mich dabei in diesem Abschnitt ausdrücklich auf die gegebene Spezifikation[37].

Bob generiert zunächst ein *Identitäts-Schlüsselpaar* IK_B, einen signierten *PreKey* SPK_B samt zugehöriger Unterschrift $Sig(IK_B, Encode(SPK_B))$ des privaten Schlüssels aus IK_B, sowie eine Reihe *Einweg-PreKeys* (OPK_B^1, OPK_B^2, OPK_B^3...) und veröffentlicht das *Bundle* (IK_B, SPK_B, Sig, OPK_B^1, OPK_B^2, OPK_B^3...) auf dem Server[37, S. 5]. Dabei werden natürlich nur die öffentlichen Schlüssel der jeweiligen Schlüsselpaare veröffentlicht. *Bob* hat damit seinen Teil getan und kann offline gehen. Anmerkung: $Encode(X)$ beschreibt eine Kodierungsfunktion, die den Schlüssel X serialisiert.

Wenn nun *Alice* eine Nachricht an *Bob* senden möchte, prüft sie zunächst, ob sie bereits früher eine Sitzung mit *Bob* erstellt hat. Falls noch keine Sitzung besteht ruft sie *Bob*'s *PreKey Bundle* vom Server ab. Dieses besteht aus seinem öffentlichen Identitätsschlüssel IK_B, dem signierten *PreKey* SPK_B, samt Signatur Sig sowie optional einem vom Server zufällig ausgewähltem Einweg-*PreKey* OPK_B.

Anmerkung: Hier unterscheidet sich die Spezifikation des OMEMO Protokolls vom Signalprotokoll: In letzterem wählt der Server einen zufälligen Schlüssel aus, sendet ihn an *Alice* und löscht diesen anschließend aus *Bob*'s *Bundle*. Wenn bereits alle *PreKeys* gelöscht wurden, sendet der Server *Alice* ein *Bundle ohne PreKey*. Im Gegensatz dazu wird im OMEMO Protokoll immer das gesamte Bundle mit allen auf dem Server vorgehaltenen *PreKeys* (OPK_B^1, OPK_B^2...) an *Alice* gesendet und diese wählt sich selbst einen *PreKey* OPK_B aus der Menge der *PreKeys*[6]. Das liegt vor allem daran, dass OMEMO serverseitig das XEP Personal Eventing Protocol (XEP-0163)[41] zur Verteilung der *Bundles* nutzt, welches keine Modifikation der veröffentlichten Daten durch den Server (etwa durch Auswahl eines bestimmten Schlüssels) erlaubt.

Als nächstes überprüft *Alice* die Signatur Sig des signierten *PreKeys* SPK_B mit Hilfe von *Bob*s Identitätsschlüssel IK_B und bricht gegebenenfalls das Ver-

fahren ab, sollte die Signatur nicht gültig sein[37, S. 5]. In dem Fall, dass das *Bundle* keinen OPK_B enthielt, generiert sie ein Einweg-Schlüsselpaar mit öffentlichem Schlüssel EK_A und führt folgende Berechnungen aus:

$$\begin{aligned} DH1 &= DH(IK_A, SPK_B) \\ DH2 &= DH(EK_A, IK_B) \\ DH3 &= DH(EK_A, SPK_B) \\ SK &= KDF(DH1 \parallel DH2 \parallel DH3) \end{aligned}$$

Sollte das *Bundle doch* einen *PreKey* OPK_B enthalten (wie es im OMEMO-Protokoll immer der Fall sein sollte), so wird zusätzlich $DH4$ errechnet und die Berechnung für SK modifiziert[37, S. 5f]:

$$\begin{aligned} DH4 &= DH(EK_A, OPK_B) \\ SK &= KDF(DH1 \parallel DH2 \parallel DH3 \parallel DH4) \end{aligned}$$

Der Ausdruck $DH(X, Y)$ beschreibt hier die Ausführung der Einwegfunktion des Diffie-Hellman Verfahrens, wie weiter zuvor beschrieben. Dabei ist X jeweils der private Schlüssel des angegebenen Paares und Y ein öffentlicher Teil.

$DH1$ und $DH2$ stellen gegenseitige Authentifizierung sicher, da *Bob* nur mit dem privaten Schlüssel von SPK_B die Möglichkeit hat, SK zu errechnen. Dieser wurde von *Bob* mit dessen Identitätsschlüssel IK_B unterschrieben. Im Idealfall haben *Alice* und *Bob* im Voraus ihre Identitätsschlüssel überprüft. $DH3$ und $DH4$ garantieren vorwärts gerichtete Sicherheit (*forward secrecy*), sowie *future secrecy* durch die Einführung neuer Zufallswerte in das Verfahren durch die Nutzung des frisch erzeugten EK_A.

Als nächstes löscht *Alice* den privaten Schlüssel von EK_A, sowie die errechneten Werte $DH1$,$DH2$,$DH3$,$DH4$, da diese nicht mehr benötigt werden und nur mehr ein Sicherheitsrisiko im Falle einer Kompromittierung darstellen. Der drei- (bzw. vier-)fachen Ausführung der DH-Funktion ist der Name des Verfahrens verschuldet.

Nun ist die Sitzung erstellt und Alice kann beginnen, die Nachricht vorzubereiten. Diese enthält folgendes[37, S. 6]:

- IK_A
- EK_A
- ID des genutzten OPK_Bs
- Zufällig generierter symmetrischer Schlüssel zur Verschlüsselung der Nachricht, wiederum verschlüsselt mit einem sogenannten AEAD[42] Verschlüsselungsverfahren mit IDs von IK_A und IK_B als AD und einem aus SK abgeleiteten Schlüssel als AE.

Sobald *Bob* die Nachricht erhält, extrahiert er *Alice' IdentityKey* IK_A (Identitätsschlüssel), sowie den öffentlichen Teil des Einwegschlüssels EK_A. Dann führt er die gleichen von Alice durchgeführten Diffie-Hellman Berechnungen ($DH1$, $DH2$, $DH3$, ($DH4$)) durch und erhält somit SK. Auch er löscht die Werte der $DH1$,$DH2$,$DH3$,($DH4$), um *forward secrecy* zu gewährleisten.

Mit SK haben nun beide Partner ein gemeinsames Geheimnis, welches zum einen zum Entschlüsseln des symmetrischen Nachrichtenschlüssels genutzt werden kann, zum anderen können beide SK nutzen, um daraus neue Schlüssel für zukünftige Nachrichten abzuleiten. Letztendlich konnte *Alice* eine Nachricht an *Bob* senden, ohne dass dieser für die Durchführung des Schlüsselaustausches online sein musste.

3.2 Double Ratchet Verfahren

Beim *Double Ratchet Verfahren* handelt es sich um einen Algorithmus, mit dem *Alice* und *Bob* eine kryptografische *Sitzung* aufbauen können, mit der für jede Nachricht neue Schlüssel generiert werden. Ein Hauptprinzip des Algorithmus ist es, Schlüssel so schnell wie möglich zu wechseln und nicht mehr verwendete zu löschen. Der Algorithmus bietet neben *Vertraulichkeit*, *Authentizität* und *Integrität* zusätzlich *forward secrecy*, *future secrecy*, sowie *plausible deniability* (vgl. 2.1.2). Durch die gleichzeitige Erfüllung der Ziele *forward-* und *future secrecy* kann man das Verfahren als *selbstheilend*[43] beschreiben. Daher stammt auch der ursprüngliche Name *Axolotl Protokoll* (der Axolotl ist ein in Mexiko lebendes Tier aus der Familie der Querzahnmolche. Er besitzt die einzigartige Fähigkeit, ganze Körperteile bei Verlust zu regenerieren[44]). In diesem Abschnitt beziehe ich mich ausdrücklich auf die Spezifikation des *Double Ratchet Verfahrens*[45].

3.2.1 Ratchet

Ein wichtiger Grundbaustein des Algorithmus ist die sogenannte *Ratchet* (wörtlich *Ratsche*). Diese beschreibt einen Mechanismus, der in etwa vergleichbar ist mit der Funktionsweise einer mechanischen Ratsche. Dabei handelt es sich um ein Werkzeug, welches ohne Widerstand in eine Richtung gedreht werden kann, jedoch beim Drehen in die andere sperrt[46]. In einem Ratchet-Algorithmus werden bei Empfang, bzw. Versand einer Nachricht neue Schlüssel mithilfe von sogenannten *Key Derivation Function Chains* (KDF-Chains, etwa *Schlüsselableitungsketten*) aus den verwendeten Schlüsseln generiert und die alten verworfen (vgl. Vorwärtsdrehung der Ratsche). Es kann so nicht aus den neuen Schlüsseln auf die alten geschlossen werden (Sperre bei Rückwärtsdrehung). Dieses sogenannte *Ratcheting* wird beim *Double Ratchet Algorithmus* an meh-

reren Stellen verwendet.

3.2.2 Key Derivation Function Chain

Eine *Key Derivation Function* (*Schlüsselableitungsfunktion*, kurz *KDF*) ist eine Funktion, die einen KDF-Schlüssel KDF_{in}, sowie eine weitere Eingabe *Input* erhält und daraus zwei neue Schlüssel KDF_{out} und *Output* generiert[45, S. 3]. In KDF-Chains wird die KDF-Funktion mehrmals hintereinander ausgeführt und KDF_{out} der vorigen Berechnung wird zum KDF_{in} der nächsten (siehe Abbildung 3.2). Die KDF-Chain verleiht dem Double Ratchet Algorithmus die Eigenschaft der *forward-* bzw. *future secrecy*[45, S. 3f].

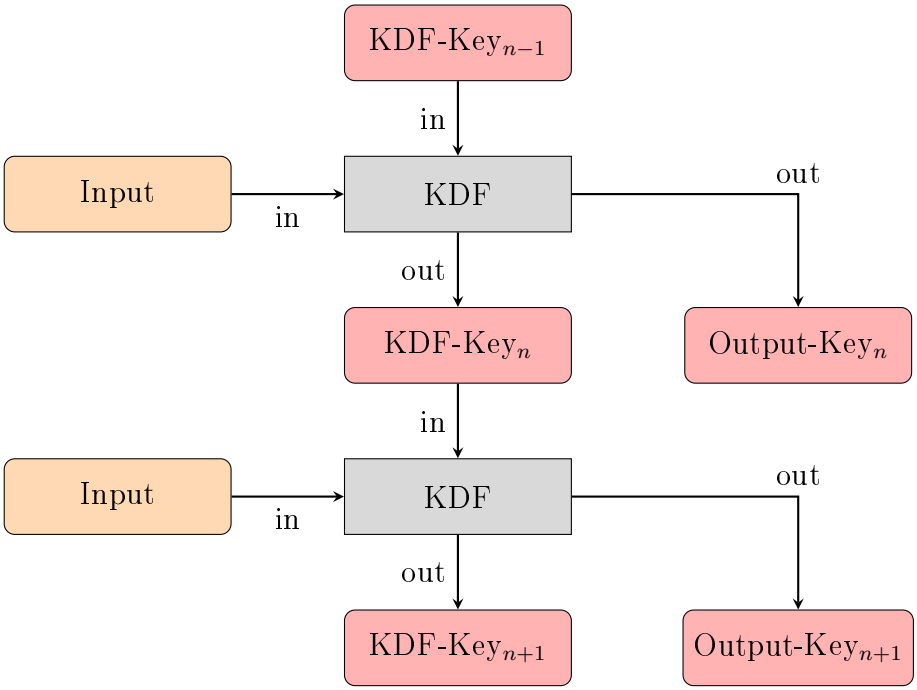

Abbildung 3.2: Funktionsweise einer Key Derivation Function Chain, eigener Entwurf nach [45].

Eine *Double Ratchet Sitzung* zwischen *Alice* und *Bob* besteht aus 3 KDF-Chains[45, S. 4]: Einer sogenannten *root chain*, einer *sending chain*, sowie einer *receiving chain*. Dabei entspricht *Alice' sending chain* der *receiving chain* von *Bob* und umgekehrt. Diese Symmetrie ermöglicht es *Alice* und *Bob* mit Hilfe ihrer *sending* bzw. *receiving chains* dieselben Schlüssel zu erzeugen wie der Kommunikationspartner.

Das *Double Ratchet*-Verfahren ist eine Kombination zweier verschiedener Ratchet-Algorithmen, nämlich der *Diffie-Hellman-Ratchet* und der *Symmetric-Key Ratchet*[45, S. 5].

3.2.3 Symmetric-Key Ratchet

Im *Symmetric-Key Ratchet* wird jede Nachricht mit einem neuen Schlüssel verschlüsselt. Die Kommunikationspartner besitzen jeweils eine *sending* und *receiving chain*, wie oben beschrieben. Wenn *Alice* eine Nachricht an *Bob* senden möchte, vollführt sie einen Schritt mit ihrer *sending chain*, um einen neuen Nachrichtenschlüssel zu erhalten. Sie verschlüsselt damit die Nachricht, sendet sie an Bob und löscht anschließend den Schlüssel. *Bob* führt mit seiner *receiving chain* die selben Schritte durch und erhält den gleichen Schlüssel zum Entschlüsseln der Nachricht. Auch er löscht den nicht mehr benötigten Schlüssel nach Verwendung. Das Verfahren erfüllt damit die Eigenschaft *forward secrecy*, dh. Nachrichten, die zum Zeitpunkt eines erfolgreichen Angriffs in der Vergangenheit liegen, bleiben sicher[45, S. 4]. Das Problem hierbei ist, dass ein Angreifer, welcher einen KDF-Key eines Kommunikationspartners stehlen kann, somit Zugriff auf alle zukünftigen Nachrichten hat (mangelnde *future secrecy*)[45, S. 6]. Abhilfe schafft die Kombination mit der *Diffie-Hellman Ratchet*.

3.2.4 Diffie-Hellman Ratchet

In diesem Algorithmus werden die Schlüssel der *sending/receiving chain* regelmäßig mit neuen Schlüsseln, welche aus einer weiteren KDF-Chain (der *root chain*) abgeleitet werden, ersetzt[45, S. 6]. Dazu generieren *Alice* und *Bob* sich ein zusätzliches DH-Schlüsselpaar (*Ratchet key*) und versenden mit jeder Nachricht ihre aktuellen öffentlichen *Ratchet keys* an den Kommunikationspartner. Empfängt ein Partner (z.B. *Alice*) einen neuen *Ratchet public key* des Anderen, so führt er einen *DH-Schlüsselaustausch* mit seinem Schlüsselpaar durch. Die Ausgabe wird in der *root chain* genutzt, um einen neuen Eingabewert für die *sending chain* zu generieren. Das eigene *Ratchet key pair* kann nun mit einem frisch generierten Schlüsselpaar ersetzt werden[45, S. 7]. Dann wird erneut ein Schlüsselaustausch zwischen empfangenem *Ratchet key* und dem neu generierten Schlüsselpaar durchgeführt, mit der Ausgabe ein weiterer *Ratchet step* der *root chain* berechnet und mit dessen Ausgabe der KDF-Key der *sending chain* ersetzt. Sollte *Alice* nun eine Antwort an *Bob* senden wollen, sendet sie ihren neuen öffentlichen *Ratchet key* mit. Somit erfüllt das Verfahren nun auch die Eigenschaft der *future secrecy*, da nach jedem *Ratchet step* neuer Zufall in Form neuer Schlüsselpaare eingeführt wird.

In Abbildung 3.3 ist ein *Ratchet step* der *Diffie-Hellman-Ratchet* beschrieben. Hier ist *priv. ratchet key*$_{A,n}$ der alte private *Ratchet key* und *priv. ratchet key*$_{A,n+1}$ der neu generierte. Den empfangenen öffentlichen *Ratchet key* des Kommunikationspartners findet man als *pub. ratchet key*$_B$ wieder. *DH* steht im Diagramm für einen Diffie-Hellman Schlüsselaustausch. Man beachte, wie ein *Ratchet step* in zwei Schritte der *root chain* resultiert[45, S. 12].

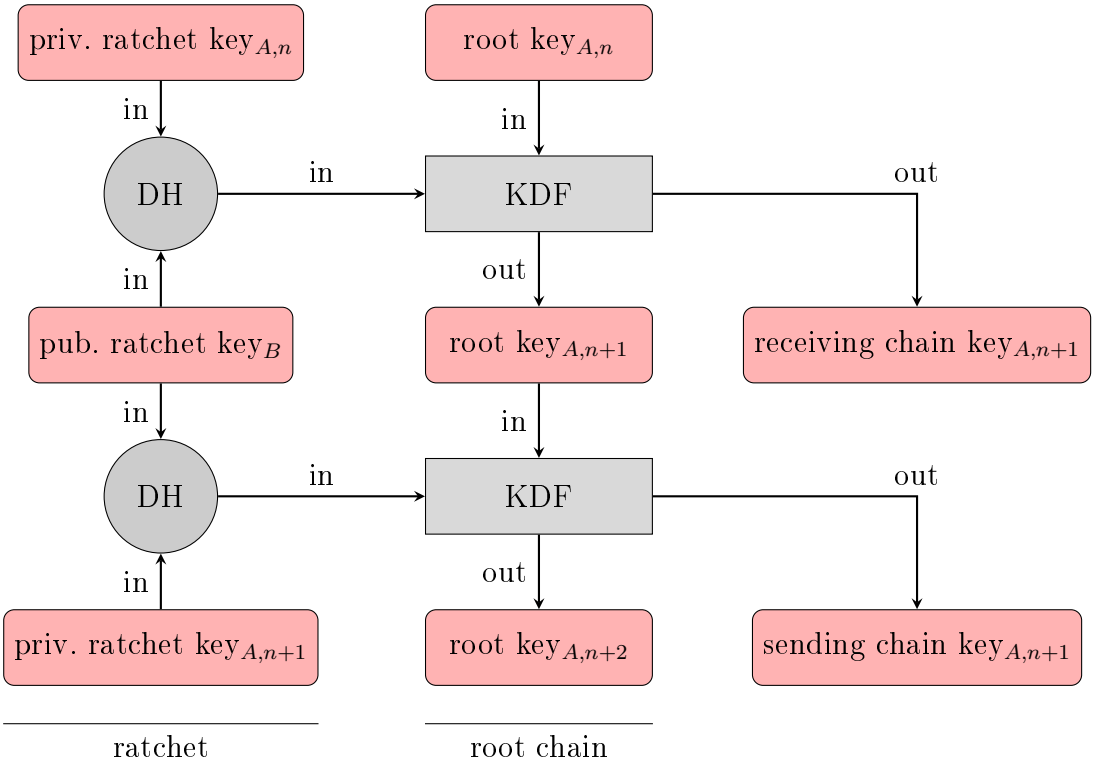

Abbildung 3.3: Funktionsweise des Diffie-Hellman Ratchet Algorithmus, eigener Entwurf nach [45].

3.2.5 Double Ratchet

Die Kombination aus *Symmetric-* und *Diffie-Hellman Ratchet* wird im Signalprotokoll als *Double Ratchet* bezeichnet[45, S. 13]. Diese vereint die Vorteile beider Verfahren, sodass sowohl *forward-*, als auch *future secrecy* gegeben sind.

Wenn Alice eine Nachricht sendet oder empfängt, vollführt sie einen *symmetric-key ratchet step* um den für die Nachricht zu nutzenden, bzw. genutzten Schlüssel abzuleiten. Wann immer sie einen neuen öffentlichen *ratchet key* von Bob empfängt, führt sie vor dem *symmetric-key ratchet step* einen *Diffie-Hellman ratchet step* mit der *root chain* aus und erstellt damit einen neuen Schlüsselbaum der *receiving/sending chain*.

Da die *root chain* immer nur beim Empfang einer Nachricht mit neuem *ratchet key* voranschreitet, kann es nicht passieren, dass durch verloren gegangene Nachrichten die Sitzung irreparabel beschädigt wird[45, S. 17]. In einer übertragenen Nachricht ist angegeben, wie viele Schritte die *sending chain* des Senders gemacht hat, bevor die Nachricht gesendet wurde und welcher Schlüssel in der *root chain* verwendet wurde. Im Fall, dass *Alice* die Nachrichten A_1, A_2, A_3 an *Bob* sendet und nur A_1 und A_3 ankommen, kann *Bob* (ggf. zu-

nächst die *root chain* voranschreiten lassen,) einen Schritt mit der *receiving chain* machen und mit dem errechneten Schlüssel A_1 entschlüsseln. Die Tatsache, dass er als nächstes A_3, anstelle von A_2 erhält sagt ihm, dass A_2 eventuell verspätet oder verloren ist. Er führt also zwei Schritte mit der *sending chain* durch und speichert den ersten Schlüssel für A_2 für den Fall, dass die Nachricht noch zu einem späteren Zeitpunkt empfangen wird. Mit dem Schlüssel für A_3 entschlüsselt er die letzte Nachricht wie gehabt und kann die Schlüssel für A_1 und A_3 löschen.

3.3 XEdDSA und VXEdDSA Signaturverfahren

Bei den XEdDSA und VXEdDSA Signatureschemata handelt es sich um Spezialfälle der EdDSA Signaturfunktionen, basierend auf X25519 und X448 Elliptic Curve Diffie Hellman Funktionen. Diese Kurven werden genutzt, da sie als besonders schnell gelten[47]. Das XEdDSA Verfahren ermöglicht es, das gleiche Schlüsselpaarformat für Signaturen und Elliptic Curve Diffie-Hellman zu nutzen. In manchen Fällen sind beide Verfahren mit dem selben Schlüsselpaar durchführbar[48, S. 2].

Das VXEdDSA Verfahren erweitert XEdDSA zu einer verifizierbaren Zufallsfunktion[48, S. 2]. Das bedeutet, dass es möglich ist, durch die Verifizierung der Signatur einen Beweis p zu erhalten, welcher garantiert eindeutig für die Nachricht und den verwendeten Schlüssel ist. Dabei kann von der Kenntnis des Beweises, sowie des öffentlichen Schlüssels nicht auf den privaten Schlüssel geschlossen werden. Somit kann ein Schlüsselpaar gleichzeitig zum Verschlüsseln und Signieren einer Nachricht genutzt werden, was den Verwaltungsaufwand verringert.

3.4 Spezifikation als XEP-0384

Hier soll genauer auf die Spezifikation von OMEMO als XMPP Extension Protocol[6] eingegangen werden.

OMEMO wurde 2015 von Andreas Straub entwickelt und für Conversations implementiert[35]. Im Herbst 2015 wurde es als Proto-XEP (noch auf Basis des Signalprotokolls) vorgeschlagen und im Dezember 2016 von der XSF als XEP-0384[6] (mit *Olm*[5] als Basis) akzeptiert.

OMEMO hat das Ziel, das mittlerweile veraltete *OTR-Verfahren*[49] als bevorzugte Verschlüsselungsmethode im Bereich der Kommunikation mittels XMPP abzulösen[6, S. 4].

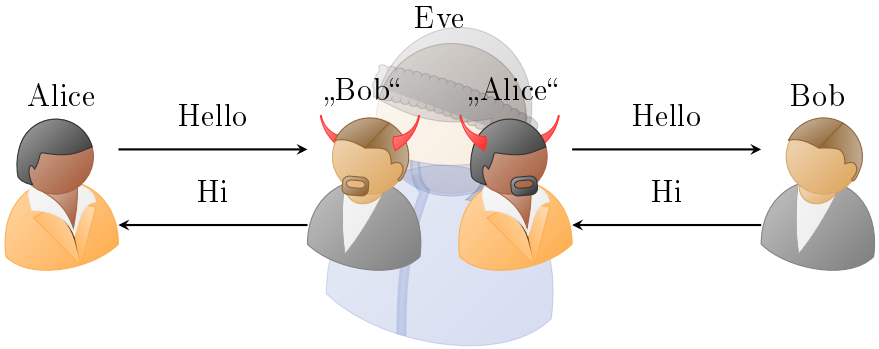

Abbildung 3.4: Schema eines Man-in-the-Middle Angriffs. Eve gibt sich gegenüber Alice als Bob und gegenüber Bob als Alice aus.

3.4.1 Audit

Die OMEMO-Spezifikation wurde im Juni 2016 einem unabhängigen Sicherheitsaudit[36] unterzogen, wobei eine Sicherheitslücke entdeckt wurde, welche jedoch bereits behoben wurde. In der anfälligen Variante war es einem kompromittierten Gerät möglich, einen Man-in-the-Middle Angriff durchzuführen (vgl. Abbildung 3.4).

Der Grund war, dass einem Angreifer, welcher sich Kontrolle über eines der genutzten Geräte beschaffen konnte, der symmetrische Schlüssel des Nachrichteninhaltes (*MessageKey*) bekannt war. Damit war es ihm möglich, den Nachrichteninhalt mit einer veränderten Nachricht auszutauschen, welche er einfach mit dem gleichen Schlüssel verschlüsseln konnte, wie die ursprüngliche Nachricht[36, S. 16] (siehe Abb. 3.5). Andere Geräte hatten in der anfälligen Variante des Protokolls keine Möglichkeit, die Authentizität der veränderten Nachricht festzustellen, da der *AES-Auth-Tag*[50] (bei Verwendung von AES-GCM) der Nachricht einfach an den verschlüsselten Inhalt angehängt wurde und somit ebenfalls der Kontrolle des Angreifers unterlag. Behoben wurde das Problem, indem der *Auth-Tag* aus dem Ciphertext-Teil in den *MessageKey* verschoben und für jeden Empfänger verschlüsselt wird. Somit ist der Angreifer nicht mehr in der Lage, den *Auth-Tag* und die Nachricht gleichzeitig zu fälschen, was den Angriff erfolgreich verhindert.

3.4.2 Voraussetzungen

Damit zwei Nutzer OMEMO-verschlüsselt kommunizieren können, müssen beide Server nutzen, die folgende XEPs implementieren[6, S. 4]:

- XEP-0163[41]: *Personal Eventing Protocol* zur Veröffentlichung und Verteilung der Schlüsselbündel.

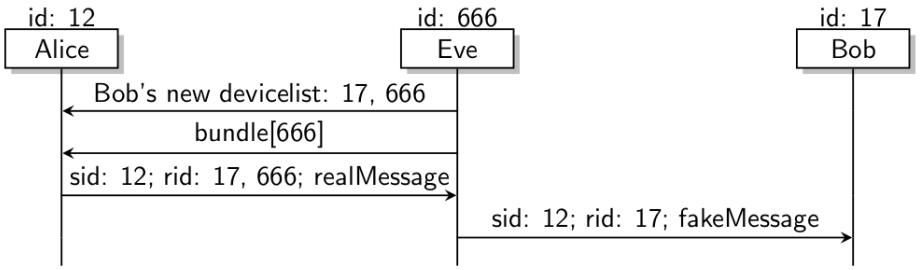

Abbildung 3.5: Beispiel eines MitM-Angriffs mit Hilfe eines kompromittierten Gerätes. Grafik aus [36]

- XEP-0280[51] (optional): *Message Carbons* zur Verteilung gesendeter Nachrichten an alle Geräte (des Senders und Empfängers).
- XEP-0313[52] (optional): *Message Archive Management* zur Speicherung von Nachrichten in einem Archiv auf dem Server. Dadurch können Geräte verpasste Nachrichten nachträglich abrufen.

Die Protokollerweiterungen *Message Carbons* und *Message Archive Management* sind dabei nicht erforderlich, aber zur vollen Ausschöpfung der durch OMEMO gegebenen Möglichkeiten empfehlenswert. Durch Ersteres wird die Zustellung empfangener Nachrichten an alle angemeldeten Geräte des Nutzers ermöglicht[51]. Auf einem Server, der dieses XEP nicht unterstützt, erhält nur die *Ressource* (Gerät des Nutzers) mit der höchsten Priorität die Nachricht. Das ist gerade im Umfeld der Mehrgerätenutzung nicht wünschenswert.

Message Archive Management hingegen ermöglicht es dem Server, Nachrichten für *Alice* zwischenzuspeichern und später an Geräte zuzustellen, die zum Zeitpunkt des Nachrichtenversands nicht angemeldet waren[52]. Auch diese Fähigkeit ist gerade bei der Nutzung mehrerer Geräte mit mitunter instabiler Internetanbindung (z.B. Mobiltelefone im Mobilfunknetz) hilfreich, um Nachrichten auf allen Geräten synchron zu halten.

3.4.3 Beispiele

Wenn Alice sich das erste mal mit einem OMEMO-fähigen Gerät anmeldet, erzeugt sie ihren Identitätsschlüssel (*IdentityKey*) und eine eindeutige Geräte-ID (*deviceId*). Als nächstes ruft sie ihre eigene Geräteliste (*deviceList*) ab, um zu überprüfen, ob weitere Geräte unter *Alice*' Nutzerkennung OMEMO-Unterstützung angemeldet haben (siehe Abbildung 3.6). Sollte sie zuvor einen neuen *IdentityKey* generiert haben, prüft sie, ob die ebenfalls frisch generierte *deviceId* bereits in der Liste vorhanden ist und erstellt ggf. eine neue. Es ist wichtig, dass die *deviceId* eindeutig ist, da darüber OMEMO-Geräte identifi-

ziert werden.

```
<message from = 'alice@wonderland.lit'
         to = 'alice@wonderland.lit'
         type = 'headline'
         id = '2QZ2r-14' >
  <event xmlns='http://jabber.org/protocol/pubsub#event'>
    <items node='urn:xmpp:omemo:0:devicelist'>
      <item>
        <list xmlns='urn:xmpp:omemo:0'>
          <device id='666338132'/>
          <device id='1723307798'/>
        </list>
      </item>
    </items>
  </event>
</message>
```

Abbildung 3.6: Eine empfangene Geräteliste

Als nächstes stellt *Alice* sicher, dass ihre eindeutige *deviceId* auf der *deviceList* vorhanden ist. Das heißt, sie fügt ihre ID gegebenenfalls zur Geräteliste hinzu und veröffentlicht die aktualisierte Liste neu (siehe Abbildung 3.7). Damit hat sie OMEMO-Unterstützung *angekündigt*.

```
<iq from='alice@wonderland.lit' type='set' id='2QZ2r-17'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='urn:xmpp:omemo:0:devicelist'>
      <item>
         <list xmlns='urn:xmpp:omemo:0'>
           <device id='666338132'/>
           <device id='1723307798'/>
           <device id='2143084793'/> <!-- <-neue Id -->
        </list>
      </item>
    </publish>
  </pubsub>
</iq>
```

Abbildung 3.7: Alice veröffentlicht eine aktualisierte Geräteliste mit ihrer deviceId

Als nächstes veröffentlicht sie gegebenenfalls ihr OMEMO *deviceBundle* mit ihrem öffentlichen *IdentityKey*, einem *SignedPreKey* samt Signatur, sowie einer Liste mit etwa 100 *PreKeys* (siehe Abbildung 3.8).

```
<iq to='alice@wonderland.lit' id='2QZ2r-25' type='set'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <publish node='urn:xmpp:omemo:0:bundles:2143084793'>
      <item>
        <bundle xmlns='urn:xmpp:omemo:0'>
          <signedPreKeyPublic signedPreKeyId='1'>
            BASE64ENCODED...
          </signedPreKeyPublic>
          <signedPreKeySignature>
            BASE64ENCODED..
          </signedPreKeySignature>
          <identityKey>
            BASE64ENCODED..
          </identityKey>
          <prekeys>
            <preKeyPublic preKeyId='1'>
              BASE64ENCODED...
            </preKeyPublic>
            <preKeyPublic preKeyId='2'>
              BASE64ENCODED...
            </preKeyPublic>
            ...
          </prekeys>
        </bundle>
      </item>
    </publish>
  </pubsub>
</iq>
```

Abbildung 3.8: Alice veröffentlicht ihr PreKeyBundle

Dieses *bundle* kann jetzt z.B. von *Bob* angefordert werden, wenn dieser verschlüsselt mit Alice kommunizieren möchte. *Alice* braucht dazu nicht online zu sein. *Bob* extrahiert die Schlüssel im *Bundle*, prüft die Signaturen und wählt sich einen zufälligen *PreKey* aus. Diesen nutzt er, um wie in Abschnitt 3.1 beschrieben um eine Sitzung mit jedem beteiligten Gerät aufzubauen. Falls bereits eine Sitzung vorhanden war, muss dieser Schritt natürlich nicht durchgeführt werden. Wenn er nun eine Nachricht verschlüsseln möchte, erstellt er zunächst einen symmetrischen Schlüssel (AES-GCM) und verschlüsselt damit den Nachrichteninhalt. Den symmetrischen Schlüssel verschlüsselt er anschließend für jedes empfangende Gerät (alle vertrauten Geräte von Alice, sowie seine anderen eigenen). Dazu nutzt er wie in Abschnitt 3.2 beschrieben die Sitzungen mit den Geräten. Die verschlüsselten *MessageKeys* werden zusammen

```
<message to='alice@wonderland.lit'
         from='bob@builder.tv'
         id='xox5u-59'
         type='chat'>
  <encrypted xmlns='urn:xmpp:omemo:0'>
    <header sid='2084601931'>    <!--Bob's sending device-->
      <key rid='666338132'>      <!--Alice's first device-->
        BASE64ENCODED MESSAGE KEY...
      </key>
      <key rid='1723307798'>     <!--Alice's second device-->
        BASE64ENCODED MESSAEG KEY...
      </key>
      <key rid='2143084793'>     <!--Alice's third device-->
        BASE64ENCODED MESSAEG KEY...
      </key>
      <key rid='6822874429'>     <!--Bob's other device-->
        BASE64ENCODED MESSAEG KEY...
      </key>
      <iv>BASE64ENCODED IV...</iv>
    </header>
    <payload>
      BASE64ENCODED ENCRYPTED MESSAGE...
    </payload>
  </encrypted>
</message>
```

Abbildung 3.9: Bob sendet eine verschlüsselte Nachricht an Alice. Die Nachricht enthält auch verschlüsselte Schlüssel für andere Geräte

mit der deviceId des jeweiligen Geräts im Header der zu sendenden Nachricht eingefügt (siehe Abbildung 3.9).

Wenn *Alice* die Nachricht empfängt, überprüft sie zunächst, ob darin ein *key* Element mit ihrer *deviceId* existiert und verwirft die Nachricht, falls kein solches Element vorhanden ist[6, S. 9]. Ansonsten lädt sie die entsprechende Sitzung, bzw. erstellt diese wie in Abschnitt 3.2 beschrieben. Dann entschlüsselt das mit ihrer *deviceId* ausgezeichnete *key*-Element, welches den verwendeten Nachrichtenschlüssel enthält. Damit kann sie anschließend die Nachricht selbst entschlüsseln, welche sich im *payload*-Element befindet. Das entschlüsselte *payload*-Element wird anschließend wie ein normales *body*-Element behandelt.

Die OMEMO-Spezifikation empfiehlt, das in der Kryptografie-Bibliothek implementierte Trustmanagement zu deaktivieren und stattdessen die Verwaltung des Schlüsselvertrauens selbst zu behandeln [6, S. 11].

3.5 Vergleich zu herkömmlicher Ende-zu-Ende Verschlüsselung

Im Bereich der Kommunikation ist Ende-zu-Ende Verschlüsselung nichts neues. Für XMPP gibt es beispielsweise u.a. zwei andere XEPs, welche Verschlüsselungsschemata spezifizieren.

3.5.1 OpenPGP

Das ältere von beiden ist XEP-0027[53], welches die Nutzung von *OpenPGP* in XMPP beschreibt. Dieses XEP stammt aus dem Jahr 2002 und wurde inzwischen vom (experimentellen) XEP-0373[54] abgelöst.

Im *OpenPGP* Verschlüsselungsverfahren besitzen beide Nutzer nur jeweils ein Schlüsselpaar. Wenn *Alice* eine Nachricht an *Bob* schreibt, besorgt sie sich zunächst *Bob*'s öffentlichen Schlüssel und nutzt diesen, um alle ihre Nachrichten an Bob zu verschlüsseln. Der Vorteil ist, dass dieses System recht übersichtlich ist. Jeder Nutzer besitzt nur ein einziges Schlüsselpaar und XMPP-Nachrichten bestehen nur aus zwei Elementen (Nachrichtensignatur und Ciphertext). Sollte ein Nutzer mehrere Geräte benutzen, so kann er sein Schlüsselpaar an diese verteilen und hat so von jedem Gerät Zugriff auf die Unterhaltung. Außerdem können auch sehr alte Nachrichten bequem entschlüsselt werden, was *OpenPGP* vor allem für Nutzer interessant macht, die gerne ein verschlüsseltes Archiv vergangener Nachrichten aufbewahren möchten. Verlorene Nachrichten, oder solche in falscher Reihenfolge haben keine Auswirkungen auf die Verschlüsselung.

Der Nachteil besteht darin, dass das Verfahren weder *forward-* noch *future secrecy* bietet. Sollte *Eve* zu beliebigem Zeitpunkt Zugriff auf *Alice*' Schlüssel erhalten, so kann sie *alle* Nachrichten, die an *Alice* gesendet wurden, mitlesen. Sollte *Alice* von dem Angriff nichts mitbekommen, so kann *Eve* außerdem auch zukünftige Nachrichten entschlüsseln. Wird *OpenPGP* ohne Signaturen verwendet, so können die Kommunikationsteilnehmer sich nicht sicher sein, mit wem sie kommunizieren. Werden hingegen Signaturen verwendet, so fehlt dem Verfahren die Eigenschaft der Abstreitbarkeit (*plausible deniability*). Marlinspike beschreibt das *OpenPGP*-Verfahren als ärchitectural dead-end"[55]. Trotz dieser Nachteile ist *OpenPGP* eine populäre Methode zum Verschlüsseln von E-Mails. Auch die Messaging-Anwendung *Kontalk* verwendet *OpenPGP* als bevorzugte Verschlüsselungsmethode[29]. Ein weiteres Anwendungsgebiet von *OpenPGP* ist das Signieren von Software-Updates (bspw. in vielen Linux Repositorien).

3.5.2 OTR

XEP-0364[49] aus dem Jahre 2015 hingegen beschreibt die verschlüsselte Kommunikation mittels OTR[9]. OTR steht für Öff the RecordMessaging und wurde vom Cypherpunks-Projekt[9] entwickelt, um die Probleme von OpenPGP zu lösen. Da OTR *Protokoll-agnostisch* ist, war eine Spezifikation als XEP lange Zeit nicht notwendig. Eingesetzt wurde OTR in Kombination mit XMPP schon lange Zeit bevor das XEP verfasst wurde. Im OTR-Verschlüsselungsverfahren werden ähnlich wie bei OMEMO für jede Nachricht neue Schlüssel erzeugt[14]. Dadurch erfüllt OTR *forward secrecy*. Des weiteren bietet OTR *plausible deniability*, da am Ende einer Sitzung die Signaturschlüssel veröffentlicht werden. Ein großer Nachteil ist, dass das Verfahren nur für die Kommunikation zwischen zwei Endpunkten eignet. Weder Konferenzen (z.B. MUCs[24]), noch die gleichzeitige Nutzung mehrerer Geräte/Ressourcen pro Nutzer sind möglich, da OTR nicht zwischen einzelnen Geräten unterscheidet. Außerdem müssen bei OTR beide Chatteilnehmer gleichzeitig online sein, da eine Sitzung *aktiv* ausgehandelt und nach beendeter Unterhaltung wieder verworfen wird.

Zur Sitzungsverhandlung wird das sogenannte *SCIMP*[56]-Protokoll genutzt[14]. Dieses wurde 2012 von Phil Zimmermanns Firma *Silent Circle* entwickelt und unter anderem in den Kommunikationsanwendungen *Silent Text*, bzw. *Silent Phone* eingesetzt. Erstere wurde im Jahr 2015 eingestellt. *Silent Phone* nutzt hingegen interessanterweise inzwischen ebenfalls den Double Ratchet Algorithmus[57].

3.5.3 OMEMO als nächste Generation?

OMEMO, bzw. das Signalprotokoll kombiniert die Vorteile beider Verfahren und löst gleichermaßen die Probleme, welche durch ein verändertes Ökosystem entstanden sind. Durch die Verwendung der *PreKeyBundles* ermöglicht OMEMO die Kommunikationsaufnahme und Sitzungserstellung auch dann, wenn eines der genutzten Endgeräte offline ist. Durch die Unterscheidung einzelner Geräte eines Nutzers wird Ende-zu-Ende verschlüsselte Kommunikation mit mehreren Endpunkten möglich. Durch die Verwendung eines IdentityKeys kann einem Nutzer ähnlich wie bei OpenPGP eine gleichbleibende Identität zugewiesen werden, die von Kontakten überprüft werden kann. Des weiteren bietet das Verfahren im Gegensatz zu OpenPGP *deniability* und *future-* sowie *forward secrecy*.

3.6 Angriffsszenarien

Betrachten wir nun einmal *Eves* Situation als Angreifer. Konträr zu den Zielen der Nutzer von OMEMO gibt mehrere Ziele, wie *Eve* die Kommunikation von *Alice* und *Bob* beeinflussen kann:

- Mitlesen verschlüsselter Nachrichten
- Fälschen/Manipulieren von Nachrichten
- Deanonymisierung der Chatteilnehmer
- Nachweis, dass Kommunikation stattgefunden hat
- Unterbindung der Kommunikation

Das OMEMO Protokoll bietet Schutzmechanismen gegen eine Reihe von Angriffen.

3.6.1 Mitlesen

Betrachten wir zunächst den wichtigsten Punkt, das Mitlesen der Kommunikation. Da *Eve* in der Kryptografie als Gegner gesehen wird, der Kontrolle über das gesamte Kommunikationsnetzwerk besitzt, gehen wir davon aus, dass sie *jede* übertragene Nachricht kopieren, speichern und verändern kann. Um eine abgefangene Nachricht entschlüsseln zu können, bräuchte *Eve* Zugriff auf verschiedene Schlüssel: Der eigentliche Nachrichtentext ist mit dem *MessageKey* verschlüsselt. Könnte *Eve* diesen kompromittieren, so wäre die Nachricht für sie lesbar. Allerdings wäre dies kein sehr lohnender Angriff, weil jede Nachricht mit einem anderen Schlüssel gesichert ist. *Eve* könnte den Angriff mit dem Schlüssel also nicht fortsetzen. Der *MessageKey* hingegen ist mit dem *RatchetKey* der *sending* bzw. *receiving chain* des Senders, bzw. Empfängers verschlüsselt. Auf diesen kann *Eve* aus abgefangenen Nachrichten nicht schließen, da mit jedem Schritt lokale Zufallswerte auf den Geräten von *Alice* und *Bob* in das Verfahren eingeführt werden, welche nicht über das Netzwerk übertragen werden. Sollte es *Eve* also zu einem Zeitpunkt schaffen, auf irgendeine Weise auf den *RatchetKey* zu schließen, so kann auch dieser Angriff nicht fortgeführt werden. Bei beiden Angriffen hat *Eve* auch keine Chance, vergangene Nachrichten zu lesen, da aus neuen Schlüsseln nicht auf alte geschlossen werden kann (vgl. 3.2).

Ein weiterer denkbarer Angriff wäre eine sogenannte *Man-in-the-Middle*-Attacke (kurz *MitM*, etwa *Mittelsmannangriff*). Dabei stellt sich *Eve* zwischen *Alice* und *Bob* und gibt sich gegenüber den beiden als jeweils der andere aus (vgl. Abbildung 3.4). Für *Alice* scheint es so, als kommuniziere sie mit *Bob*, *Bob* gegenüber gibt sich *Eve* hingegen als *Alice* aus. Solche *MitM*-Angriffe sind sehr effektiv und können im schlimmsten Fall das sicherste Verschlüsselungs-

verfahren überwinden, da keine Kryptografie besiegt werden muss.

Als Gegenmaßnahme bietet OMEMO die Möglichkeit, *IdentityKeys* über einen externen Kanal zu verifizieren. Dazu wird beispielsweise der *Fingerprint* (SHA1-Prüfsumme des *IdentityKeys*) angezeigt, welcher vom Nutzer verglichen werden kann (z.B. durch Scannen eines QR-Codes). Der Hintergrund ist, dass *Eve* nicht in der Lage ist, einen *IdentityKey* mit beliebiger Prüfsumme zu generieren.

3.6.2 Fälschen/Manipulieren von Nachrichten

Es könnte ein lohnendes Ziel für *Eve* sein, Nachrichten zwischen *Alice* und *Bob* zu manipulieren. Sie könnte beispielsweise den Nachrichteninhalt abändern wollen, um so Falschinformationen zu verbreiten. Dagegen ist das OMEMO Protokoll gefeit, indem die Nachrichtenschlüssel und auch der *Auth-Tag* verschlüsselt sind. Letzterer garantiert, dass die Nachricht auf dem Weg vom Sender zum Empfänger nicht verändert wurde. Sollte *Eve* den Inhalt manipuliert haben, schlägt die Überprüfung des *Auth-Tags* fehl und der Empfänger wird auf die Manipulation aufmerksam. An dieser Stelle war das Protokoll in einer frühen Version angreifbar (vgl. 3.4.1).

Um Nachrichten fälschen, bzw. erfolgreich manipulieren zu können, müsste *Eve* den *Auth-Tag* der Nachricht fälschen. Das ist im Normalfall nicht möglich, da *Eve* dazu an Schlüsselmaterial gelangen müsste, welches nie über das Netzwerk übertragen wird.

3.6.3 Unterbinden der Kommunikation

Um die Kommunikation zwischen *Alice* und *Bob* zu verhindern, könnte *Eve* beispielsweise verhindern, dass die Kommunikationspartner in den Besitz der OmemoBundles des jeweils anderen kommen. Auf diese Weise könnten *Alice* und *Bob* keine kryptografischen Sitzungen miteinander aufbauen. Genauso gut könnte sie jedoch auch einfach die genutzten Server direkt angreifen, um jegliche Kommunikation zu verhindern (DDoS). In einem solchen Fall könnten *Alice* und *Bob* sich jedoch einfach neue Server suchen und ihre Kommunikation dort fortsetzen. Dies wird durch die dezentrale Natur des XMPP-Netzwerkes ermöglicht. Es sei des weiteren angemerkt, dass das OMEMO-Protokoll auch dann noch funktioniert, wenn einzelne Nachrichten verloren gehen oder in falscher Reihenfolge beim Empfänger ankommen.

3.7 Mögliche Weiterentwicklung

OMEMO hat das Potential, OTR und OpenPGP im Bereich des Instantmessaging abzulösen. Jedoch gibt es ein paar Punkte, die ich eventuell für zukünftige Entwicklungen in Erwägung ziehen würde.

Zum einen nutzt OMEMO kein Web-of-Trust, wie es z.B. aus OpenPGP bekannt ist. Es wäre vielleicht eine Überlegung wert, IdentityKeys bei Verifizierung zu signieren und die Signaturen zu veröffentlichen, sodass Nutzer bei Kontaktaufnahme einen ersten groben Eindruck bekommen können, ob der Kontakt vertrauenswürdig ist oder nicht. Natürlich wirft das weitere Fragen zur Privatsphäre auf, da jeder sehen könnte, wer mit wem Kontakt hat.

Alternativ könnte man die IdentityKeys seiner Kontakte auch nur für sich selbst signieren und in einen privaten PubSub-Konten auf dem Server ablegen. Damit könnte man eventuell das Vertrauen in die Geräte von Kontakten auf den eigenen Geräten synchronisieren. Momentan muss man bei Nutzung eines neuen Gerätes jedes mal neu entscheiden, ob man den Geräten eines Kontakts vertraut oder nicht. Analog muss man jedes mal, wenn man einem Gerät vertraut oder misstraut, die gleiche Änderung auf allen anderen Geräten vornehmen. Das kann mitunter umständlich und unübersichtlich werden. Wenn jedes Gerät die IdentityKeys vertrauter Geräte signieren würde, müsste ein neues Gerät A bloß einem älteren Gerät B vertrauen und vertraut damit automatisch transitiv allen Geräten, denen auch B vertraut:

$$A \xrightarrow{\text{vertraut}} B \xrightarrow{\text{vertraut}} C$$
$$\Rightarrow$$
$$A \xrightarrow{\text{vertraut}} C$$

Letztere Idee könnte man zum Beispiel im Bereich des *IoT* (*Internet of Things*) nutzen. In der XSF gibt es eine Arbeitsgruppe, die sich mit der Nutzung von XMPP im Kontext von IoT auseinander setzt und untersucht, inwieweit XMPP zur sicheren und einheitlichen Kommunikation von *Things* beitragen kann[58].

IoT-Geräte verfügen oftmals über kein komfortables Interface zur Konfiguration. Ergo wäre auch die Verwaltung von Vertrauen eine schwierige Aufgabe. Das könnte man lösen, indem man ein *Master*-Gerät einrichtet, welches entscheidet, welche Geräte vertrauenswürdig sind. Jedes *Thing* müsste dann lediglich dem *Master* vertrauen, und könnte dessen Vertrauensentscheidungen übernehmen. Auf diese Art wäre es auch möglich, *Things* zu Gruppen zusammen zu fassen. Jedes *Thing* dürfte dann bspw. nur anderen *Things* vertrauen, mit denen es eine Gruppe teilt.

4 Implementation und Ergebnisse

4.1 Ziele und Aufgaben

Ziel meiner Implementation ist die Umsetzung des im XEP-0384[6] spezifizierten Protokolls für die *Smack*-Bibliothek[2] mithilfe der Kryptografiebibliothek *libsignal-protocol-java*[1]. Dies umfasste folgende Aufgaben:

- OMEMO-spezifische Stanzas interpretieren und erstellen
- Auf eingehende Stanzas reagieren
- Schaffung einer Infrastruktur zur Schlüsselverwaltung
- Nutzung der *libsignal-protocol-java* für kryptografische Operationen

Die Ergebnisse meiner Arbeit sind auf Github[59, 60] zu finden.

Im Zentrum der Implementation steht die Klasse *OmemoManager*, welche dem Nutzer grundsätzliche Methoden zur OMEMO Verschlüsselung zur Verfügung stellt. Die Klasse besitzt unter anderem Methoden zum Verschlüsseln von Nachrichten, zum Registrieren eines *OmemoMessageReceivedListeners*, welcher entschlüsselte empfangene Nachrichten an den Nutzer der Bibliothek zurück liefert, sowie zum Regenerieren der eigenen Identitätsschlüssel und zum Entfernen anderer Geräte aus der *DeviceList*.

Die Smack Bibliothek stellt sogenannte *Provider* bereit, mit deren Hilfe eingehende XML-*Stanza*s in *ExtensionElement*-Objekte umgewandelt werden. Für meine Implementation habe ich meine eigenen Provider-Klassen *OmemoBundleProvider*, *OmemoDeviceListProvider*, sowie *OmemoMessageProvider* entworfen, welche OMEMO-spezifische Stanzas erkennen und automatisch in Objekte der entsprechenden Java-Klassen umwandeln.

Der *OmemoStore* ist für die Speicherung von Schlüsseln, sowie weiterer Informationen, wie dem Vertrauensstatus zuständig. Wie die Daten genau gespeichert werden (Datenbank, einzelne Dateien o.ä.) lasse ich dabei dem Nutzer übrig. Definiert werden lediglich die Interfaces der Methoden.

Da ein *Bundle*, wie im XEP definiert[6], eine andere Form hat, als ein *PreKeyBundle* (libsignal-protocol-java[1]), (es enthält alle PreKeys des Nutzers, nicht nur einen) muss meine Implementation zufällig einen PreKey aus-

wählen, und damit ein neues *PreKeyBundle* erstellen, welches an die Signal-Bibliothek weitergegeben wird.

Eine Nachricht wird vom *OmemoService* mit Hilfe eines *OmemoMessage-Builder*s erstellt. Diese Klasse kümmert sich darum, den Inhalt einer Nachricht mit einem AES Schlüssel (*MessageKey*) zu verschlüsseln und diesen anschließend mit Hilfe der *OmemoSession*s der Empfängergeräte für jedes Gerät zu chiffrieren.

Eingehende *OmemoMessage*s werden zunächst vom *OmemoMessageProvider* geparst und anschließend von einem internen Listener an den *OmemoService* weitergegeben. Dieser prüft, ob die Nachricht einen Schlüssel enthält, der mit der eigenen *deviceId* gekennzeichnet ist. Dieser wird mit der *OmemoSession* des Senders entschlüsselt und bei Erfolg wird der entschlüsselte AES Schlüssel genutzt um an den Inhalt der Nachricht zu kommen, welcher als Klartext-*Message* an den registrierten *OmemoMessageReceivedListener*, bzw. *OmemoMucMessageReceivedListener* zurückgegeben wird.

4.2 Kurzüberblick die Implementation

Im Folgenden gebe ich einen kurzen Überblick über das Ergebnis meiner Implementation des OMEMO-Protokolls als Erweiterung für die Java-basierte XMPP-Bibliothek *Smack* mithilfe von *libsignal-protocol-java* als Kryptografiebibliothek.

Meine Implementation unterstützt sowohl Einzel- als auch Gruppenkonversationen, sowie den Versand von Offline-Nachrichten. Außerdem werden *MessageCarbons* unterstützt, dh. Nachrichten werden zwischen allen genutzten Geräten synchronisiert. Des weiteren bietet meine Implementation eine Gegenmaßnahme für die in Abschnitt 3.4.1 genannte Schwachstelle. Diese kann (noch) zu Kompatibilitätszwecken über einen Wahrheitswert ein- und ausgeschaltet werden.

Was meine Implementation noch nicht bietet ist ein funktionales Vertrauensmanagement. Standardmäßig vertraut meine Implementation jeder Identität.

Die Implementation ist aus Gründen der Modularität in zwei Module aufgeteilt. Das ermöglicht die einfache Portierung der Erweiterung auf eine alternative Verschlüsselungsbibliothek (z.B. *Olm*).

4.2.1 smack-omemo

Das Modul *smack-omemo*[59] enthält, wie weiter oben bereits beschrieben, von verwendeten kryptografischen Bibliotheken unabhängige Klassen, sowie Interfaces und abstrakte Klassen, welche von konkreten Implementationen imple-

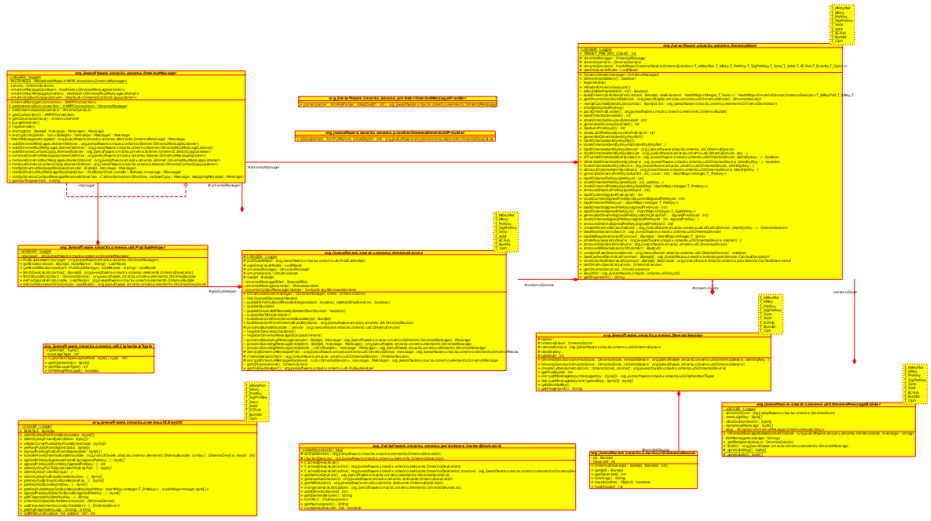

Abbildung 4.1: UML-Diagramm der wichtigsten Klassen des Moduls *smack-omemo*

mentiert werden müssen (siehe Abbildung 4.1). Zu letzteren gehören vor allem der *OmemoStore*, welcher die persistente Speicherung von Schlüsseln (z.B. in einer Datenbank) umsetzen soll. Der Programmierer, welcher das Modul in seiner Anwendung nutzen möchte, muss diese Klasse erweitern und Methoden zur Speicherung und zum Laden verschiedener Daten umsetzen. Empfohlen wird die Nutzung einer verschlüsselten Datenbank (z.B. *sqlcipher*[61]) zur sicheren Ablage der kryptografischen Schlüssel im Speicher.

Außerdem finden sich in diesem Modul sogenannte Providerklassen, die eingehende XMPP-Stanzas in Java-Objekte parsen.

4.2.2 smack-omemo-signal

Das Modul *smack-omemo-signal*[60] ist eine konkrete Implementation des Moduls *smack-omemo*[59] mithilfe der Bibliothek *libsignal-protocol-java*[1]. Da *libsignal-protocol-java* unter der GPL-Lizenz steht[8], *Smack* hingegen unter der *Apache Public License* lizenziert ist[7], war die Aufspaltung in zwei Module notwendig, um die eventuelle spätere Integration von *smack-omemo* mit einer anderen konkreten Implementation (z.B. mit *Olm*[5]) in das Smack-Projekt zu ermöglichen, ohne eine Lizenzänderung vornehmen zu müssen. Der Grund hierfür ist die virale Eigenschaft der GNU General Public License[62], welche die Nutzung von GPL-Code in APL-Programmen verhindert[63].

Es sei angemerkt, dass das Modul *smack-omemo-signal* aus nur 5 Klassen besteht (siehe Abbildung 4.2). Eine Portierung auf eine andere Kryptografiebibliothek sollte also relativ schnell und einfach umzusetzen sein.

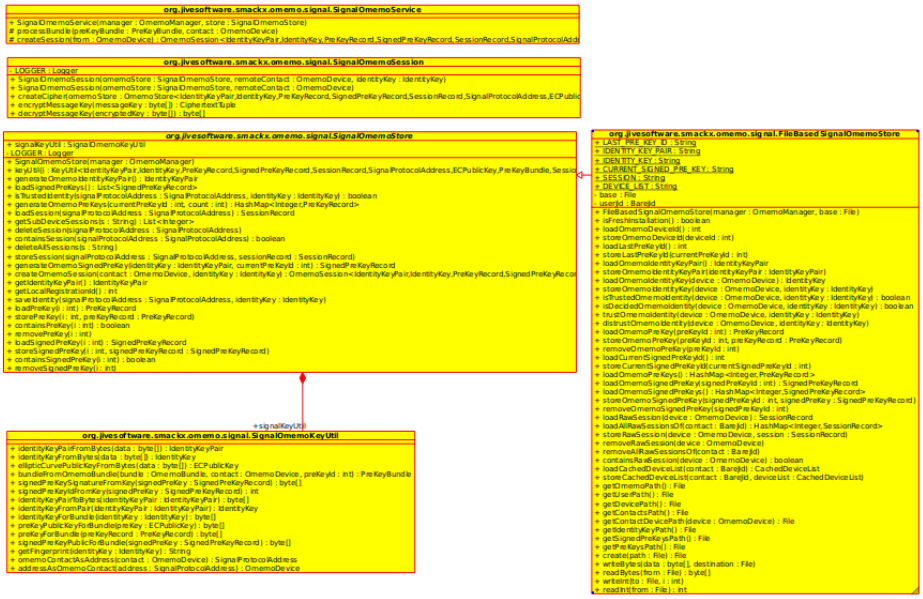

Abbildung 4.2: UML-Diagramm der wichtigsten Klassen des Moduls *smack-omemo-signal*

Die Klassen im Modul besitzen die gleichen Namen wie ihre Gegenstücke im *smack-omemo*-Modul, nur jeweils mit dem Prefix *Signal*. Ich empfehle für eine spätere Olm-Implementation dementsprechend den Prefix *Olm* zu verwenden.

4.2.3 Beispiele

Wenn *Alice* die beiden Module in ihrem Programm nutzen möchte, erstellt sie zunächst einen *OmemoManager*, sowie eine konkrete Implementation des *SignalOmemoStore*s. Der *OmemoStore* lädt bestehende Schlüssel, bzw. generiert neue, sollten keine bestehenden Schlüssel gefunden werden. Hier kann beispielsweise der *FileBasedSignalOmemoStore* verwendet werden. Zusätzlich wird ein konkretes Objekt des *SignalOmemoService*s benötigt, welches den erstellten *OmemoManager* und *OmemoStore* übergeben bekommt. Anschließend sollte der Nutzer noch Listener für OMEMO-Nachrichten in Einzel- und Gruppenchats registrieren. Eine Beispielimplementation kann so aussehen (Abbildung 4.3):

Intern werden beim Start die aktuelle *DeviceList* des Nutzers abgerufen, und - im Falle, dass die eigene ID noch nicht dabei ist, diese hinzugefügt und die Liste neu veröffentlicht. Außerdem wird ein *OmemoBundle* erstellt und veröffentlicht, welches es anderen Nutzern ermöglicht, mit *Alice* verschlüsselten Kontakt aufzunehmen.

Wenn *Alice* nun eine Nachricht an *Bob* (hier die Grinsekatze) verschlüsseln

möchte, ruft sie die Methode *encrypt(BareJid bobsJid, Message message)* auf und erhält, gegebenenfalls, ein verschlüsseltes *Message* Objekt, welches wie eine normale Nachricht versendet, und von *Bob*s Geräten, die OMEMO unterstützen und in der Geräte-Liste waren, entschlüsselt werden kann (Abbildung 4.4).

Wenn *Bob* (hier die Grinsekatze) eine Nachricht an *Alice* sendet, wird diese vom Modul erkannt, welches automatisch versucht, die Nachricht zu entschlüsseln. *Alice* bekommt den Klartext bei erfolgreicher Entschlüsselung, durch den *OmemoMessageReceivedListener* zurückgeliefert.

Der ab hier folgende Code ist nicht notwendig, da er automatisch aufgerufen wird, jedoch verdeutlicht er sehr gut die interne Funktionsweise (Abbildung 4.5).

Falls die Nachricht nicht entschlüsselt werden konnte (z.B. mangels passendem Schlüssel), wird sie stillschweigend verworfen.

Um zu bestimmen, welche Geräte eines Kontakts OMEMO unterstützen, ruft *Alice* dessen *Device List PEP-Node* ab (Abbildung 4.6). Falls die Node existiert, unterstützt der Kontakt OMEMO auf allen Geräten, deren *DeviceIDs* in der zurückgelieferten Liste enthalten sind.

Auf gleiche Weise kann *Alice* nun das *OmemoBundle* eines der Geräte in der Liste abrufen (Abbildung 4.7):

Dieses enthält neben dem öffentlichen *IdentityKey* des Nutzers, einen *SignedPreKey* mit Signatur, sowie eine Liste von 100 sogenannter *PreKeys*. Aus diesen kann *Alice* sich zufällig einen auswählen und damit eine Sitzung erstellen, mit der Zukünftige Nachrichten ver- und entschlüsselt werden können. Die kryptografischen Hintergründe und Verfahren können Kapitel 2 entnommen werden.

```
// connect and login
connection = new XMPPTCPConnection("alice@wonderland.lit",
"password", "wonderland.lit").connect();
connection.login();

// OMEMO
OmemoManager omemoManager = OmemoManager
.getInstanceFor(connection);

FileBasedSignalOmemoStore omemoStore =
new FileBasedSignalOmemoStore(omemoManager,
new File("/home/alice/chat/store"));

new SignalOmemoService(omemoManager, store);

//Listeners
OmemoMessageListener messageListener = (bareJid, message) -> {
if(bareJid != null && message != null) {
System.out.println(bareJid+": "+message.getBody());
}
};

OmemoMucMessageListener mucMessageListener =
(multiUserChat, bareJid, message) -> {
if(multiUserChat != null && bareJid != null
&& message != null) {
System.out.println(multiUserChat.getRoom()+":
"+bareJid+": "+message.getBody());
}
};

omemoManager.addOmemoMessageListener(messageListener);
omemoManager.addOmemoMucMessageListener(mucMessageListener);

//enable message carbons
CarbonManager.getInstanceFor(connection).enableCarbons();
```

Abbildung 4.3: Simpler Chat Client

```
BareJid recipient = JidCreate.bareFrom(
"chesirecat@wonderland.lit");
Message message = new Message();
message.setBody("But I don't want to go among mad people.");
Message secret = omemoManager.encrypt(recipient, message);
chatManager.createChat(recipient.asEntityJidIfPossible())
.sendMessage(secret);
```

Abbildung 4.4: Senden einer verschlüsselten Nachricht

```
Message secret;
// Some method to receive the message
Message clear = omemoService.decrypt(chesire, secret);
omemoMessageListener.onOmemoMessageReceived(clear);

Output: Chesirecat:
Oh, you can't help that, we're all mad here. I'm mad. You're mad.
```

Abbildung 4.5: Veranschaulichung der Entschlüsselung

```
OmemoDeviceList deviceList = omemoService.getPubSubHelper()
.fetchDeviceList(bobsJid);
```

Abbildung 4.6: Abrufen einer DeviceList

```
int deviceId = deviceList.get(0);
OmemoContact omemoContact = new OmemoContact(bobsJid, deviceId);
OmemoBundle bundle = omemoService.getPubSubHelper()
.fetchBundle(omemoContact);
```

Abbildung 4.7: Abrufen eines OmemoBundles

5 Fazit

In den folgenden Abschnitten werde ich kurz erläutern, auf welche unerwarteten Hindernisse ich gestoßen bin, bzw. welche Erwartungen erfüllt, bzw. nicht erfüllt wurden.

5.1 Fazit zum theoretischen Teil

Während der Recherchephase meiner Arbeit habe ich viel über die im Signalprotokoll verwendeten Mechanismen gelernt und konnte so mein Verständnis der Kryptografie erweitern. Ich habe gelernt, Protokollspezifikationen zu lesen und zu interpretieren.

5.2 Fazit zum Praxisanteil

Generell habe ich den Aufwand der Schlüsselverwaltung unterschätzt. Ich habe erfahren müssen, dass Kryptografie das Problem der Geheimhaltung versandter Nachrichten in eines der Schlüsselverwaltung verwandelt.

Kleinere Schwierigkeiten hat mir die spärliche Dokumentation der *libsignal-protocol-java*-Bibliothek bereitet. Hier wahr häufig ein Blick in den Quelltext notwendig. Sehr hilfreich war an vielen Stellen die OMEMO-Referenzimplementation in *Conversations*, an der ich mich in einigen (vor allem sicherheitskritischen) Punkten orientieren konnte.

Eine Herausforderung war es, den Code durch die intensive Nutzung von *Generics* modular und erweiterbar zu halten. Darunter hat in erster Linie die Lesbarkeit des Quelltextes gelitten.

Bei meiner Arbeit bin ich auf ein Problem mit *Prosody* Servern gestoßen. Deren *PEP*-Implementation ist teilweise inkompatibel mit *Smack*, daher musste ich einen Workaround für das Abrufen von *PubSub*-Nodes anbringen.

Mein Ziel, Nachrichten empfangen und senden zu können habe ich erreicht. Zur vernünftigen Nutzung des Quellcodes müsste ein Chatclient zusätzlich noch Vertrauensmanagement implementieren und den Nutzer ggf. dazu auffordern, Vertrauensentscheidungen zu treffen.

Abbildungsverzeichnis

Literaturverzeichnis

[1] OpenWhisperSystems. signal-protocol-java https://github.com/whispersystems/libsignal-protocol-java. Accessed: 2016-12-21.

[2] Smack Java XMPP Library https://github.com/igniterealtime/Smack/. Accessed: 2016-12-21.

[3] Daniel Gultsch. Conversations https://conversations.im/. Accessed: 2016-12-21.

[4] Gajim plugin gajim-omemo https://dev.gajim.org/gajim/gajim-plugins/wikis/OmemoGajimPlugin. Accessed: 2016-12-21.

[5] Olm https://matrix.org/git/olm/about. Accessed: 2016-12-21.

[6] Andreas Straub. XEP-0384: OMEMO Encryption https://xmpp.org/extensions/xep-0384.pdf 2016. Accessed: 2016-12-21.

[7] License Smack https://github.com/igniterealtime/Smack/blob/master/LICENSE. Accessed: 2016-21-12.

[8] License libsignal-protocol-java https://github.com/WhisperSystems/libsignal-protocol-java/blob/master/LICENSE. Accessed: 2016-12-21.

[9] Off-the-Record Messaging https://otr.cypherpunks.ca/. Accessed: 2016-12-22.

[10] Ronald L. Rivest, Adi Shamir, Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems 1978.

[11] D. Dolev, A. Yao. *O*n the security of public key protocols . 1983.

[12] Wikipedia - Kryptographie https://de.wikipedia.org/wiki/Kryptographie#Ziele_der_Kryptographie. Accessed: 2016-12-21.

[13] Isaak Crofton. *C*rypto Anarchy.

[14] cypherpunks.ca. Off-the-Record Messaging Protocol version 3 https://otr.cypherpunks.ca/Protocol-v3-4.1.1.html. Accessed: 2017-02-12.

[15] Stephan Fischer, Christoph Rensing, Utz Rödig. *O*pen Internet Security: Von den Grundlagen zu den Anwendungen.

[16] Wikipedia - Asymmetrisches Kryptosystem https://de.wikipedia.org/wiki/Asymmetrisches_Kryptosystem. Accessed: 2017-01-08.

[17] Wikipedia - Symmetrisches Kryptosystem `https://de.wikipedia.org/wiki/Symmetrisches_Kryptosystem`. Accessed: 2017-01-08.

[18] XMPP RCF 6120 `https://tools.ietf.org/html/rfc6120`. Accessed: 2016-12-21.

[19] XMPP RCF 6121 `https://tools.ietf.org/html/rfc6121`. Accessed: 2016-12-21.

[20] XMPP RCF 6122 `https://tools.ietf.org/html/rfc6122`. Accessed: 2016-12-21.

[21] XMPP.org. XMPP | An Overview of XMPP `https://xmpp.org/about/technology-overview.html`. Accessed: 2017-01-22.

[22] Peter Saint-Andre, Kevin Smith, Remko Tronçon. *X*MPP: The Definitive Guide . 2009.

[23] XMPP Standards Foundation `https://xmpp.org/about/xmpp-standards-foundation.html`. Accessed: 2017-01-08.

[24] XEP-0045 - Multi-User-Chat `https://xmpp.org/extensions/xep-0045.html`. Accessed: 2016-12-21.

[25] XMPP.org. XMPP | XMPP Libraries `https://xmpp.org/software/libraries.html`. Accessed: 2017-01-22.

[26] Redsolutions. Xabber - Smack based Android messenger `https://github.com/redsolution/xabber-android`. Accessed: 2016-12-21.

[27] yaxim - Smack based Android messenger `https://yaxim.org/`. Accessed: 2016-12-21.

[28] GuardianProject. ChatSecure - Smack based Android messenger `https://guardianproject.info/apps/chatsecure/`. Accessed: 2016-12-21.

[29] Kontalk - Smack based messenger for Android `https://kontalk.org/`. Accessed: 2016-12-21.

[30] anderScore. anderChat - Smack based Multimessenger `https://www.anderscore.com/en/services/anderchat/`. Accessed: 2016-12-21.

[31] Jitsi - Smack based Multimessenger `https://jitsi.org/`. Accessed: 2016-12-21.

[32] Zom - Smack based Android messenger `https://zom.im/`. Accessed: 2016-12-21.

[33] Spark - Smack based crossplatform messenger `https://www.igniterealtime.org/projects/spark/`. Accessed: 2016-12-21.

[34] Wikipedia: Signal-Protokoll `https://de.wikipedia.org/wiki/Signal-Protokoll`. Accessed: 2016-12-22.

[35] Wikipedia: OMEMO https://de.wikipedia.org/wiki/OMEMO. Accessed: 2016-12-22.

[36] Sebastian Verschoor. OMEMO:CRYPTOGRAPHIC ANALYSIS REPORT https://conversations.im/omemo/audit.pdf. Accessed: 2016-12-21.

[37] Moxie Marlinspike, Trevor Perrin. The X3DH Key Agreement Protocol https://whispersystems.org/docs/specifications/x3dh/ 2016. Accessed: 2016-12-21.

[38] Whitfield Diffie, Martin E. Hellman. New Directions in Cryptography https://www-ee.stanford.edu/~hellman/publications/24.pdf 1976.

[39] Wikipedia - Diskreter Logarithmus https://de.wikipedia.org/wiki/Diskreter_Logarithmus. Accessed: 2017-01-08.

[40] Victor S. Miller. *U*se of Elliptic Curves in Cryptography . 1986.

[41] Peter Saint-Andre, Kevin Smith. XEP-0163: Personal Eventing Protocol https://xmpp.org/extensions/xep-0163.pdf. Accessed: 2017-01-08.

[42] Phillip Rogaway. Authenticated-Encryption with Associated-Data http://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf 2002. Accessed: 2016-12-23.

[43] Moxie Marlinspike. Advanced Cryptographic Ratcheting https://whispersystems.org/blog/advanced-ratcheting/ 2013.

[44] Wikipedia - Axolotl https://de.wikipedia.org/wiki/Axolotl. Accessed: 2017-01-08.

[45] Trevor Perrin, Moxie Marlinspike. The Double Ratched Algorithm https://whispersystems.org/docs/specifications/doubleratchet/ 2016. Accessed: 2016-12-21.

[46] Wikipedia - Ratsche https://de.wikipedia.org/wiki/Ratsche. Accessed: 2017-01-08.

[47] Daniel J. Bernstein. Curve 25519: New Diffie Hellman Speed Records https://cr.yp.to/ecdh/curve25519-20060209.pdf.

[48] Trevor Perrin. The XEdDSA and VXEdDSA Signature Schemes https://whispersystems.org/docs/specifications/xeddsa/ 2016. Accessed: 2016-12-21.

[49] Sam Whited. XEP-0364 - Current Off-the-Record Messaging Usage https://xmpp.org/extensions/xep-0364.pdf. Accessed: 2017-01-08.

[50] Wikipedia: Galois/Counter Mode https://de.wikipedia.org/wiki/Galois/Counter_Mode. Accessed: 2017-02-10.

[51] Joe Hildebrand, Matthew Miller. XEP-0280: Message Carbons https://xmpp.org/extensions/xep-0280.pdf. Accessed: 2017-01-08.

[52] Matthew Wild, Kevin Smith. XEP-0313: Message Archive Management https://xmpp.org/extensions/xep-0313.pdf. Accessed: 2017-01-08.

[53] XEP-0027: Current Jabber OpenPGP Usage https://xmpp.org/extensions/xep-0027.html. Accessed: 2017-01-17.

[54] XEP-0373: OpenPGP for XMPP https://xmpp.org/extensions/xep-0373.html. Accessed: 2017-01-30.

[55] Moxie Marlinspike. Youtube: Next Generation Threats https://www.youtube.com/watch?v=tOMiAeRwpPA 2014.

[56] (Vinnie Moscaritolo), Gary Belvin, Phil Zimmermann. SCIMP: Silent Circle Instant Messaging Protocol Specification https://cdn.netzpolitik.org/wp-upload/SCIMP-paper.pdf 2012.

[57] Silent Circle. What is Silent Phone? https://web.archive.org/web/20160304091202/https://support.silentcircle.com/customer/en/portal/articles/2118686-what-is-silent-phone-. Accessed: 2017-02-08.

[58] Peter Saint-Andre, Rikard Strid. XEP-0381 Internet of Things Special Interest Group (IoT SIG) https://xmpp.org/extensions/xep-0381.pdf. Accessed: 2017-02-21.

[59] Paul Moritz Schaub. smack-omemo Github-Reposity https://github.com/vanitasvitae/smack-omemo.

[60] Paul Moritz Schaub. smack-omemo-signal Github-Reposity https://github.com/vanitasvitae/smack-omemo-signal.

[61] GuardianProject. SQLCipher - Encrypted Database https://guardianproject.info/code/sqlcipher/.

[62] Free Software Foundation. GNU General Public License V3 http://www.gnu.de/documents/gpl.de.html. Accessed: 2017-01-06.

[63] Wikipedia - Viral License https://en.wikipedia.org/wiki/Viral_license. Accessed: 2017-01-06.

[64] Daniel Gultsch. OMEMO Übersicht https://conversations.im/omemo/. Accessed: 2016-12-21.

[65] Daniel J. Bernstein, Simon Josefsson, Tanja Lange, Peter Schwabe, Bo-Yin Yang. EdDSA for more curves https://ed25519.cr.yp.to/eddsa-20150704.pdf 2015.

[66] Wikipedia - Extensible Messaging and Presence Protocol https://de.wikipedia.org/wiki/Extensible_Messaging_and_Presence_Protocol. Accessed: 2017-01-08.