Bachelorarbeit geschrieben im Rahmen der Bachelorprüfung
im Studiengang Informatik
am Institut für Informatik

des Fachbereichs Mathematik und Informatik

Ende-zu-Ende-Verschlüsselung mit OMEMO für das Kommunikationsprotokoll XMPP. Analyse anhand einer Implementierung für *libpurple*

Germán Márquez Mejía
marquez.mejia@fu-berlin.de

Betreuer: Prof. Dr. Matthias Wählisch
Zweitgutachter: Prof. Dr. Jochen Schiller

Berlin, 17. Mai 2017

Zusammenfassung

Anhand der Prototypimplementierung eines Plugins für die Sofortnachrichtenbibliothek *libpurple* zur Ende-zu-Ende-Verschlüsselung von Eins-zu-Eins-Nachrichten wird die experimentelle XMPP-Protokollerweiterung OMEMO (XEP-0384) evaluiert, um Verbesserungsmöglichkeiten bei der Spezifikation zu identifizieren und potenzielle Probleme zwecks Förderung des Standards zu korrigieren.

Inhaltsverzeichnis

1 Einleitung

Das Extensible Messaging and Presence Protocol (XMPP) ist ein Client-Server Protokoll für Echtzeitkommunikation, das vor allem bei Sofortnachrichtendiensten eingesetzt wird. XMPP ist nicht nur das Protokoll hinter WhatsApp [1] und Google Talk [2], sondern auch Bestandteil eines großen Netzwerks unabhängiger und miteinander kompatibler Dienste, die ähnlich wie die E-Mail-Dienste interagieren können. Es gibt zahlreiche Server und Clients, die mindestens die Kerneigenschaften unterstützen. Viele von ihnen implementieren auch optionale Protokollerweiterungen.

Obwohl XMPP strenge Authentifizierungsmechanismen (z. B. SASL) und die Sicherung der Kommunikationskanäle (z. B. mit TLS) vorsieht, verlangen die aktuellsten Sicherheitsstandards Verschlüsselungsansätze, die die Daten komplett vom Sender bis zum Empfänger schützen. Für diese Ende-zu-Ende-Verschlüsselung (E2E) wurden kryptografische Algorithmen entwickelt, die den Zugriff auf den Inhalt der Nachrichten durch einen Angreifer verhindern, sogar wenn dieser die Server kontrolliert oder freien Zugriff auf die übertragenen Daten hat. Ein solcher Algorithmus ist der Double-Ratchet-Algorithmus, der die Basis für die XMPP-Protokollerweiterung OMEMO darstellt.

OMEMO ist ein Versuch, eine einheitliche und klar definierte Prozedur zu spezifizieren, die E2E-Verschlüsselung *innerhalb* von XMPP ermöglicht und standardisiert, ohne dabei die Präsenzorientierung und die asynchrone Natur des Protokolls auszuschließen. Wir werden diese Protokollerweiterung analysieren und in Form eines Prototyps implementieren, um folgende Beiträge zu leisten:

- Mitgestaltung des Standards durch Korrektur und Verbesserung der Spezifikation
- Verbesserung der Client-Interoperabilität durch Aufdeckung von Fehlern und Inkompatibilitäten bei bestehenden Implementierungen
- Bestimmung empfehlenswerter Praxisansätze bei neuen OMEMO-Implementierungen

Im nächsten Abschnitt führen wir die Methoden ein, die bisher die E2E-Verschlüsselung in XMPP ermöglichen sowie die Schwächen, die durch OMEMO ausgeglichen werden sollen. In Abschnitt 3 erklären wir die Grundlagen von XMPP und einer nützlichen Erweiterung, die von OMEMO vorausgesetzt wird. Danach schildern wir in Abschnitt 4 die OMEMO-Spezifikation, sowohl auf XMPP-Ebene als auch auf kryptografischer Ebene. In Abschnitt 5 setzen wir diese Spezifikation anhand einer Prototypimplementierung als Plugin für die *libpurple*-Bibliothek um, und untersuchen kurz dessen Skalierbarkeit in Abschnitt 6. Zuletzt identifizieren wir einige Probleme mit der Spezifikation, die wir während der Ausarbeitung gefunden haben und bieten diesbezüglich Verbesserungsvorschläge an.

2 Bisherige Lösungsansätze

Um die Vertraulichkeit, Integrität und Authentizität der Kommunikation in XMPP zu gewährleisten, werden bisher OTR oder OpenPGP als Lösungen für E2E-Verschlüsselung eingesetzt.

OpenPGP [3] ist ein asymmetrisches kryptografisches Verfahren, das besonders in der E-Mail-Kommunikation eingesetzt wird. Es war die erste Methode zur E2E-Verschlüsselung in XMPP und die gängigen Implementierungen richten sich nach einer *de facto* Vereinbarung [4]. Seit Neuestem ist allerdings eine vollständige Lösung standardisiert [5, 6], die die alte ersetzen soll und XMPP-eigene Methoden wie PEP (siehe Abschnitt 3.3) zur Schlüsselverteilung benutzt. Die Optionen zur Schlüsselverifizierung sind die gleichen wie im normalen OpenPGP. Die asynchrone Kommunikation, d. h. der Nachrichtenaustausch zwischen Endpunkten, die nicht gleichzeitig online sind, ist dem Verfahren angehaftet. Durch die Nutzung eines einzelnen Schlüsselpaares für alle XMPP-Clients eines Benutzers ist es gleichgültig, auf welchem Client dieser eine eingehende Nachricht zu entschlüsseln versucht.

Der größte Nachteil von OpenPGP ist, dass alle Nachrichten mit einem einzelnen Schlüssel verschlüsselt werden. Ist dieser Schlüssel einmal kompromittiert, ist der gesamte Nachrichtenverkehr nachträglich entschlüsselbar. Die sog. *forward secrecy* ist deshalb mit OpenPGP nicht garantiert. Andere Nachteile dieses Modells sind die relativ schwierige Konfiguration und das Problem der Verteilung des privaten Schlüssels zwischen den Clients eines Nutzers. Der vorgeschlagene Standard bietet z. B. eine Lösung an, bei der der private Schlüssel verschlüsselt auf einem XMPP-Server gespeichert ist, was eine gewisse Vertrauensbasis zwischen Client und Server voraussetzt. Diese Annahme trifft aber in der Realität nicht immer zu.

Das sog. ***O**ff-**t**he-**R**ecord Messaging* Protokoll (OTR) wurde als Alternative zu OpenPGP entworfen, um die o. g. Probleme zu beheben und andere kryptografische Eigenschaften hinzuzufügen, die bei Sofortnachrichtenkommunikation ebenfalls wünschenswert sein können, nämlich die Abstreitbarkeit der Nachrichten (*repudiation*) und die Glaubhaftigkeit des Abstreitens (*plausible deniability*). So wie OpenPGP verwendet OTR langlebige Schlüsselpaare. Diese werden aber nur bei digitalen Signaturen eingesetzt, also um die Authentizität der Daten zu garantieren. Die tatsächliche Verschlüsselung der Daten erfolgt mit kurzlebigen Schlüsseln, die bei jedem Nachrichtenaustausch asymmetrisch vereinbart werden, sodass die Kompromittierung eines dieser Schlüssel nur eine zeitlich eingeschränkte Verletzung der Vertraulichkeit (*confidentiality*) der Kommunikation bedeutet. Durch Nutzung von Nachrichtenauthentifizierungscodes (MAC) wird die Abstreitbarkeit erreicht, ohne die Authentizität zu opfern [7].

Wenngleich OTR eine wesentliche Verbesserung der E2E-Verschlüsselung bezüglich der kryptografischen Eigenschaften im Vergleich zu OpenPGP bedeutet, wirkt sich diese Verbesserung nachteilig auf die Benutzbarkeit im XMPP-Modell aus. OTR arbeitet auf einer höheren Protokollebene und verwendet andere Echtzeitkommunikationsprotokolle nur als Transporte. Deshalb ist es kein Teil von XMPP und richtet sich nicht nach der XMPP-Logik. Dies äußert sich in zwei aus der Sicht

von XMPP großen Nachteilen: die Kommunikation muss synchron stattfinden (d. h. beide Endpunkte müssen gleichzeitig online sein) und jede Nachricht muss an einen einzelnen empfangenden Endpunkt gerichtet sein, der im Voraus festgelegt ist (nur ein Endpunk [Client] pro Empfänger). Wie wir später sehen werden, sind diese Merkmale von OTR inkompatibel mit grundlegenden XMPP-Eigenschaften.

		OpenPGP	OTR	OMEMO
Krypto	Authentizität	✓	✓	✓
	forward secrecy	✗	✓	✓
	Abstreitbarkeit	✗	✓	✓
XMPP	mehrere Endpunkte	✓	✗	✓
	asynchrone Komm.	✓	✗	✓

Tabelle 1 – Eigenschaften der unterschiedlichen E2E-Methoden in XMPP. Vertraulichkeit und Integrität werden von sämtlichen Methoden gewährleistet.

Wie in Tabelle 1 dargestellt, ist die Motivation von OMEMO, die Lücken beider bisherigen Lösungsansätze zu schließen, sodass außer den bestehenden kryptografischen Eigenschaften von OTR auch die XMPP-Kompatibilität von OpenPGP berücksichtigt wird.

3 XMPP

In diesem Abschnitt geben wir einen Überblick über die für OMEMO relevanten Eigenschaften von XMPP. Manche Konzepte sind zwecks Verständlichkeit abstrakt dargestellt, sodass sie an verschiedenen Stellen der tatsächlichen Spezifikation nicht vollständig entsprechen. Für weitere Details über die XMPP-Architektur siehe [8].

3.1 Definition und Funktionsweise

Das ***Ex**tensible **M**essage and **P**resence **P**rotocol* (XMPP) gehört zur Familie der Netzprotokolle auf der Anwendungsschicht über TCP-Verbindungen mit Client-Server-Architektur, wie z. B. SMTP oder HTTP. Clients benutzen Server als Anbindungspunkte für die Kommunikation mit anderen Clients. Darüber hinaus sind die Server direkt miteinander verbunden (siehe Abbildung 1).

Die Hauptmerkmale des Protokolls sind:

- XMPP ist *föderiert*, d. h. das XMPP-Netz besteht zwar aus Servern und Clients in unterschiedlichen Domains, aber die Kommunikation zwischen ihnen ist domainübergreifend transparent.

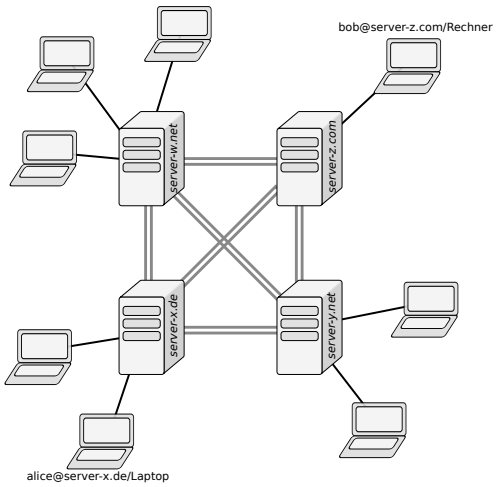


Abbildung 1 – Architektur eines XMPP-Netzwerks

- XMPP ist *präsenzorientiert*, d. h. der Verfügbarkeitszustand jedes Kommunikationspartners wirkt auf den Datenverkehr ein, sodass davon abhängige Entscheidungen getroffen werden müssen, wenn Anwendungsfälle spezifiziert werden.
- XMPP ist *erweiterbar*, d. h. die Kernprotokollspezifikation bleibt minimal und beinhaltet nur Vorgaben zur Interaktion mit unteren Netzwerkschichten und die Datenflussbausteine. Übrige Funktionalitäten sind optional und werden getrennt spezifiziert.
- XMPP zielt auf *Echtzeit-Kommunikation*, d. h. es wurde von Anfang an so konzipiert, dass die übertragenen Daten desto relevanter sind, je kürzer die Verzögerung ist, mit der sie den Empfänger erreichen. Dies hat als Folge, dass XMPP besonders oft bei Sofortnachrichtendiensten oder IoT eingesetzt wird.

Die Daten in einem XMPP-Netz werden auf Basis von XML-Streams, sei es direkt über TCP oder indirekt über HTTP mit BOSH [9], verteilt. Die Dateneinheiten auf Protokollebene sind XML-Elemente, die sog. *Stanzas*. Es gibt drei solche Elemente: *Nachricht-Stanzas* (`<message/>`), *Präsenz-Stanzas* (`<presence/>`) und *Information-/Abfrage-Stanzas* (kurz *IQ*) (`<iq/>`). Die ersten zwei sind Einweg-Stanzas. Sie dürfen unaufgefordert versendet und empfangen werden, ohne auf eine Antwort warten bzw. eine Antwort liefern zu müssen. Der Umgang mit Präsenz-Stanzas richtet sich i. d. R. nach dem Beobachter-Muster. IQ-Stanzas dagegen arbeiten nach dem Abfrage-Antwort-Prinzip und können vom Typ *get*, *set*, *result* oder *error* sein, sodass zu jeder IQ-Stanza vom Typ *get* oder *set*, die in eine Richtung fließt, eine entsprechende Antwort in Form einer IQ-Stanza vom Typ *result* oder *error* in die Gegenrichtung gehört. Alle Stanzas werden u. a. mit einem eindeutigen ID-Attribut versehen.

XMPP-Clients sind *Kommunikationsendpunkte*. Mehrere Kommunikationsendpunkte können einem Kommunikationsteilnehmer zugeordnet sein, solange sie zur gleichen Domain gehören. Ein *Kommunikationsteilnehmer* ist ein Benutzer (z. B. Alice), der gleichzeitig an mehreren Clients mit verschiedenen Netzwerksverfügbarkeitsgraden vertreten sein kann (z. B. online am Mobiltelefon und unsichtbar am PC). Kommunikationsteilnehmer sowie Kommunikationsendpunkte werden durch Adressen identifiziert. Eine XMPP-Adresse heißt *Jabber-ID* (kurz *JID*). Die drei wichtigsten Teile einer JID sind Benutzername, Domain und Ressource (siehe Abbildung 2). Der *Benutzername* hat die gleiche Aufgabe wie bei E-Mail. Die *Domain* entspricht dem Server, auf dem das Konto registriert ist. Zusammen sind sie ausreichend, um einen Kommunikationsteilnehmer im XMPP-Netzwerk eindeutig zu identifizieren und ihm Stanzas zu senden. Wenn *bob@server-z.com* eine Stanza an *alice@server-x.de* verschickt, geht die Stanza von Bobs Client nach *server-z.com*. Nach einem DNS-Lookup baut *server-z.com* eine Verbindung mit *server-x.de* auf. Während der TLS-Vereinbarung dieser Verbindung verifizieren sich beide Server gegenseitig anhand ihrer Zertifikate. Die Stanza wird an *server-x.de* weitergeleitet und dieser weiß dann, dass sie an einen seines Benutzers namens *alice* zuzustellen ist, sobald sie an einem Kommunikationsendpunkt verfügbar ist.

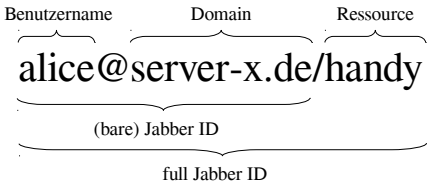


Abbildung 2 – Struktur einer XMPP-Adresse

Um einen spezifischen Kommunikationsendpunkt zu adressieren, kann eine XMPP-Adresse zusätzlich um eine sog. *Ressource* erweitert werden. Wenn eine JID die Ressource enthält, spricht man von einer *vollständigen JID* (*full Jabber ID*). Eine an eine vollständige JID adressierte Stanza wird prinzipiell nur an den entsprechenden Kommunikationsendpunkt zugestellt. Enthält die Stanza die bloße JID (engl. *bare JID*), also ohne Ressource, wird sie an alle zurzeit verfügbaren Kommunikationsendpunkte zugestellt bzw. an den ersten, der verfügbar wird. Man kann Abweichungen von diesem Verhalten beobachten, wenn Erweiterungen wie Message Carbons (XEP-0280 [10]) oder Message Archive Management (XEP-0313 [11]) vorhanden sind. Das XEP-0280 modifiziert das Kernprotokoll so, dass Nachrichten alle Kommunikationsendpunkte erreichen, selbst wenn diese weder direkt noch indirekt adressiert wurden. XEP-0313 verursacht, dass Nachrichten auch Kommunikationsendpunkte erreichen, die erst später verfügbar werden, nicht nur den ersten.

Obwohl die vollständige JID hinreichend für die eindeutige Identifizierung eines Kommunikationsendpunktes ist, gibt es keine Garantie dafür, dass sie konstant bleibt. Bei jeder Client-Server Verbindung kann der Kommunikationsendpunkt

einen beliebigen Ressourcennamen annehmen.

3.2 Erweiterbarkeit

Den Inhalt der Stanzas und dessen Semantik und Bearbeitungsregeln bestimmen die ***X**MPP **E**xtension **P**rotocols* (XEP). Ein XEP ist eine Menge von Vorgaben, die zusammen Eigenschaften von XMPP standarisieren, die neue Funktionen möglich machen. XMPP-Stanzas sind relativ inhaltsunabhängig (*payload agnostic*). Diese Tatsache verleiht dem Protokoll ein großes Erweiterungspotential. Dafür aber müssen Clients und Server klaren Standards folgen, um dialogfähig zu bleiben. Ein typisches XEP hat u. a. folgende Eigenschaften:

- Es führt spezielle Begrifflichkeiten ein, die die Beschreibung der Spezifikation vereinfachen und eindeutig machen.
- Es definiert ein XML-Schema, das neue Kindelemente samt Namensräume für die Stanzas definiert. Diese dienen der Serialisierung der zu übertragenden Daten und der Signalisierung der zu übernehmenden Vorgängen.
- Es erklärt die Semantik der neuen Elemente, wo sie vorkommen dürfen oder nicht und ggf. wie sie verschickt werden sollen.
- Es beschreibt die Verarbeitung der Stanzas in den unterschiedlichen Anwendungsfällen, d. h. es schreibt vor, was mit ihnen zu tun ist, was nicht getan werden darf, was untersagt oder optional ist.
- Es schlägt Lösungsansätze für eventuelle Probleme vor sowie Vorgehensweisen, Voraussetzungen und Invarianten.

Die Vorgaben und Definitionen im XEP dürfen sowohl den Client als auch den Server betreffen, sodass Implementierungen eventuell Software für beide Rollen in der Protokollarchitektur bereitstellen müssen. Ebenso dürfen sie andere XEPs voraussetzen oder empfehlen und auf ihnen basieren.

Das XMPP-Kernprotokoll wird von der Internet Engineering Task Force (IETF) standarisiert [12]. Die Einrichtung, die die XMPP-Erweiterungen kontrolliert, fördert und genehmigt, ist die XMPP Standards Foundation (XSF), deren Ausschuss und Mitglieder über die Aufnahme und Einstufung neuer XEPs entscheiden.

Der Weg von der Aufnahme eines XEP seitens der XSF bis hin zu seiner Standardisierung, besteht hauptsächlich aus drei Phasen: *experimental*, *draft* und *final* (für andere mögliche Phasen siehe [13]). Im experimentellen Zustand sind Prototypimplementierungen zu erstellen, die Verbesserungsmöglichkeiten, Fehler oder Lücken im XEP finden. Diese können potenziell zu radikalen Modifizierungen der Spezifikation führen. Im Entwurfszustand dürfen wesentliche Änderungen weiterhin auftauchen, jedoch ist der Einsatz in Produktionsumgebungen nicht mehr untersagt. Wenn aber ein XEP als endgültig eingestuft wird, sind nur minimale Änderungen erlaubt und Implementierungen werden für stabil und sicher gehalten.

3.3 Personal Eventing Protocol

Das ***P**ersonal **E**venting **P**rotocol* (PEP) ist eine XMPP-Protokollerweiterung, die einfache Informationsverteilung zwischen Kommunikationsteilnehmern möglich macht und im XEP-0163 [14] spezifiziert wird. Wenn ihr Server dies unterstützt, kann Alice beliebig viele sog. PEP-Knoten hochladen, die durch einen XML-Namensraum identifiziert sind. Ein *PEP-Knoten* besteht aus einem oder mehreren Items. Ein *Item* beinhaltet im Namensraum definierten XML-Elemente.

Die Funktionsweise von PEP richtet sich nach dem Beobachter-Muster (*Pubsub*). Die Knoten sind die Subjekte und die Kommunikationsteilnehmer sind die Beobachter. Es existiert zusätzlich die Möglichkeit, Knoten explizit abzufragen.

Während Alice den Inhalt der Knoten bereitstellt, übernimmt der Server dessen Verteilung. Jeder Kommunikationsteilnehmer, der Alice's Präsenz empfangen darf, darf auch Knotenbenachrichtigungen abonnieren bzw. Knoten explizit abfragen. Alice's Server liefert den Knoteninhalt an diese Teilnehmer in drei Fällen, wie in Abbildung 3 dargestellt.

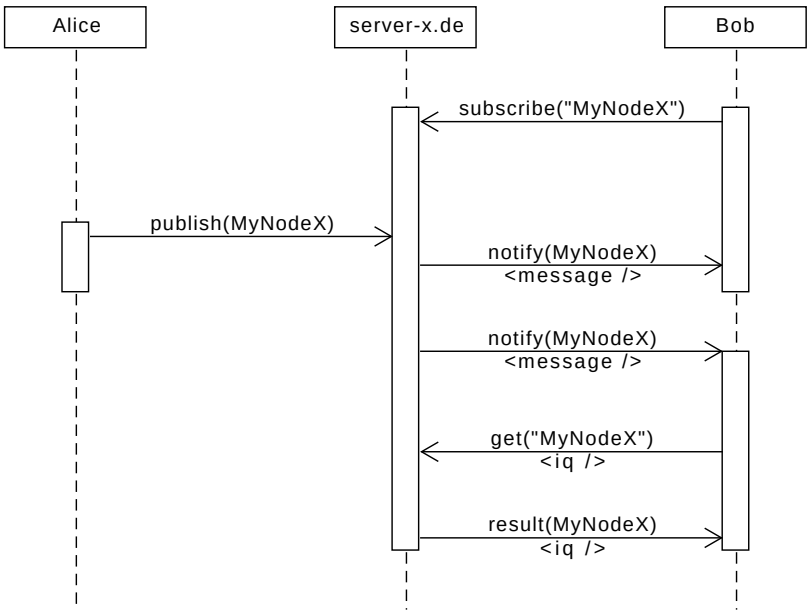


Abbildung 3 – Varianten der Inhaltsverteilung über PEP. Bobs Server, der als Proxy zwischen Bob und *server-x.de* agieren sollte, ist einfachheitshalber nicht dargestellt.

- Wenn ein Teilnehmer den Knoten anhand seines Namens abonniert hat, bekommt er unaufgeforderten Benachrichtigungen, sobald der Knoten aktualisiert wird, d. h. wenn Alice neuen Inhalt hoch lädt (evtl. überschreibt) oder

Items löscht. Solche Benachrichtigungen kommen in Form von Ereignissen innerhalb von Nachricht-Stanzas.

- Benachrichtigungen werden auch wie oben jedes Mal verschickt, wenn ein Abonnent auf einem seiner Kommunikationsendpunkten verfügbar wird (z. B. online geht), obwohl der Knoten sich nicht geändert hat.
- Letztlich kann ein Teilnehmer (Abonnent oder nicht) einen Knoten explizit abfragen. In diesem Fall werden sowohl die Abfrage als auch die Antwort mit den Items in IQ-Stanzas (jeweils `<iq type="get"/>` und `<iq type="result"/>`) transportiert.

Dies funktioniert auch, wenn Alice auf allen seiner Endpunkten offline ist, sodass sie PEP-Knoten als entfernter Datenspeicher für Informationen nutzen kann, die sie öffentlich machen möchte und die unabhängig von ihrem Verfügbarkeitszustand von anderen aufrufbar sein sollen.

4 OMEMO

XMPP unterstützt zwar sichere Client-zu-Server- und Server-zu-Server-Verbindungen, sodass die Vertraulichkeit der übertragenen Daten zwischen Kommunikationsendpunkten bewahrt wird, solange die Server selbst nicht kompromittiert sind und *alle* Verbindungen auf dem Zustellweg abgesichert sind. Doch es gibt keine Garantie dafür. Alice hat nur Einfluss auf die Sicherheit der Verbindung zwischen ihrem Client und ihrem Server, nicht darüber hinaus. OMEMO stellt deshalb die E2E-Verschlüsselung für XMPP bereit, damit die Vertraulichkeit der Daten unabhängig sowohl von der Verbindungsabsicherung als auch vom Serververtrauen ist.

OMEMO ist ein rekursives Akronym für ***O**MEMO **M**ulti-**E**nd **M**essage and **O**bject Encryption* und sein Ziel ist es, präsenzbewusste E2E-Verschlüsselung mit *forward secrecy* in XMPP zu standardisieren und, so wie XMPP, mehrere Kommunikationsendpunkte je Kommunikationsteilnehmer zu unterstützen. Es wird im XEP-0384 [15] spezifiziert, welches sich zur Zeit des Schreibens im experimentellen Zustand befindet und ausschließlich Vorgaben für den Client beinhaltet. Von Servern setzt das XEP nur PEP-Unterstützung voraus. Solange PEP vorhanden ist, benötigt dann eine neue OMEMO-Implementierung lediglich clientseitige Software.

4.1 Allgemeine Konzepte

Obwohl XMPP nicht nur bei Sofortnachrichtendiensten eingesetzt wird, bleibt dieses Anwendungsgebiet die traditionelle Beispielquelle, wenn es um neue XEPs und Anwendungsfallbeschreibungen geht. Nächstfolgend wird dieser Ansatz ebenfalls gewählt, wenngleich mit Vorbehalt, denn die Kommunikation muss nicht zwangsläufig im Rahmen von Chat-Unterhaltungen oder gar zwischen Personen stattfinden.

In der OMEMO-Spezifikation sowie im Kontext von Sofortnachrichtendiensten werden Kommunikationsteilnehmer als *Kontakte* bezeichnet. Alice's Kontakte sind

z. B. Bob, ihre Mutter und ihre Freundin Christina. Ebenso wie im XMPP-Kern, werden Kontakte anhand der bloßen JID identifiziert. Bob ist z. B. an *bob@server-z.com* zu adressieren und Christina an *chris85@server-x.de*. Jeder Kommunikationsendpunkt wird *Gerät* genannt, jedoch anders als im XMPP-Kern, ist OMEMO nicht auf die vollständige JID zur Gerät-Identifizierung angewiesen. Wie in Abschnitt 3.1 angeführt sind vollständige JIDs nicht persistent, sodass ein Wechsel des Ressourcennamens immer möglich ist (*alice@server-x.de/handy* kann plötzlich *alice@server-x.de/mobile* werden). Dies würde aber eine OMEMO-Sitzung zerstören. Stattdessen werden Geräte in OMEMO anhand von Tupeln identifiziert, die aus bloßer JID und Gerät-ID bestehen. Eine *Gerät-ID* ist eine ganze Zahl zwischen 1 und $2^{31} - 1$. Diese muss eindeutig unter den Geräten eines bestimmten Kontaktes sein und zur Installationszeit des Clients zufällig generiert werden.

Geräte werden dann als die Enden betrachtet, zwischen denen die Verschlüsselung besteht. Die Beziehung zwischen zwei Geräten auf OMEMO-Ebene wird Sitzung genannt. Eine *Sitzung* ist eine persistente Datenstruktur, die die nötigen Elemente für einen verschlüsselten Nachrichtenaustausch beinhaltet. Die Sitzungsdauer ist unabhängig vom Verfügbarkeitszustand der Geräte und von den versendeten oder empfangenen Nachrichten. Sie kann so lang sein wie die Lebensdauer der Geräte und wird nur nach der expliziten Mitteilung eines Teilnehmers beendet (z. B. aus Sicherheitsgründen), um unmittelbar danach eine neue Sitzung aufzubauen.

4.2 Elemente im OMEMO-Namensraum

Das OMEMO XML-Schema definiert drei Elemente, die für das Serialisieren der zu übertragenden Daten dienen. Diese Elemente bilden den Inhalt der PEP-Knoten und der verschlüsselten Nachricht-Stanzas. Darüber hinaus stimmen sie mit wichtigen abstrakten Begriffen überein, die für den Datenaustausch relevant sind.

4.2.1 Geräteliste

Eine *Geräteliste* ist eine Sammlung von Gerät-IDs. In dieser Liste sind die IDs sämtlicher Geräte von Alice. Jedes Gerät trägt sich selbst ein, um zu signalisieren, das Nachrichten an Alice auch für ihn verschlüsselt werden sollten. Die Liste befindet sich in einem PEP-Knoten mit einem wohl bekannten Namen mit einem einzelnen Item, dessen Kindelement `<list/>` ist.

Um sich einzutragen, ruft ein Gerät zuerst die Geräteliste auf. Sollte es keinen solchen Knoten im PEP-Datenspeicher vorhanden sein, erzeugt es eine neue Liste mit seiner ID drin. Existiert bereits der Knoten, veröffentlicht das Gerät eine neue Geräteliste, die die alte überschreibt und sowohl die vorhandenen Gerät-IDs als auch seine eigene beinhaltet. Die Geräte von Alice un Bob werden daraufhin über die Modifizierung benachrichtigt.

4.2.2 Bündel

Ein *Bündel* ist die serialisierte Form der nötigen öffentlichen Schlüssel, die ein Gerät im Voraus über seinen eigenen PEP-Knoten veröffentlicht. Der Name des Knoten besteht aus einem wohl bekannten Präfix gefolgt von der Gerät-ID. Dies bedeutet, das Alice so viele Bündel-Knoten auf ihrem PEP-Datenspeicher besitzt, wie Geräte sie hat. Diese PEP-Knoten sind nur dank der Geräteliste auffindbar, da sonst die ID-Suffixe unbekannt wären.

Ein Bündel wird als `<bundle/>` serialisiert. Dieses Element hat wiederum Kinder, die die Schlüsseldaten beinhalten:

- *<identity-key/>* behält den öffentlichen Teil eines Identitätsschlüsselpaares. Dies dient zur Authentifizierung gegenüber anderen Geräten. Die Lebensdauer des Identitätsschlüssels entspricht der Lebensdauer des Geräts.
- *<signed-pre-key-public/>* behält den öffentlichen Teil eines Schlüsselpaares, der mit dem privaten Identitätsschlüssel signiert wurde (ein sog. *signierter* Vorschlüssel). Dies wird u. a. zur Ableitung eines gemeinsamen Schlüssels mit anderen Geräten verwendet.
- *<signed-pre-key-signature/>* behält die Signatur des signierten Vorschlüssels.
- *<pre-keys/>* behält mehrere `<pre-key>`-Elemente, die jeweils den öffentlichen Teil eines Schlüsselpaares beinhalten (sog. Vorschlüssel), dessen Aufgabe ähnlich wie die vom signierten Vorschlüssel ist.

4.2.3 OMEMO-Element

Ein *OMEMO-Element* ist ein XML-Element, das verschlüsselte Inhalte serialisiert und innerhalb von Nachricht-Stanzas vorkommen darf. Es besteht aus einem `<encrypted>`-Element mit einem `<header/>` und, optional, einem `<payload/>`. Im `<header/>` befindet sich eins oder mehr sog. *OMEMO-Umschläge*, die als `<key/>` serialisiert werden (siehe Auflistung 1).

Auflistung 1 – Grundstruktur eines OMEMO-Elements

```
<encrypted>
        <header>
                <key/>
                <!--...-->
                <key/>
        </header>
        <payload/> <!--optional-->
</encrypted>
```

Abhängig davon, ob ein `<payload/>` vorhanden ist, kann ein OMEMO-Element zwei unterschiedliche Aufgaben erfüllen:

- Mit `<payload/>` ist das OMEMO-Element ein sog. *Nachricht-Element* und entspricht nach der Verschlüsselung dem `<body/>` einer Nachricht-Stanza. Der Klartext resultiert aus der Entschlüsselung von `<payload/>` anhand von den Entschlüsselungsdaten in einem Umschlag. Weiteres dazu wird in späteren Abschnitten erklärt.
- Ohne `<payload/>` ist das OMEMO-Element ein sog. *Schlüsseltransport-Element*. Nach der Verschlüsselung entsteht kein Inhalt sondern nur die Entschlüsselungsdaten, die aus einem Umschlag extrahiert werden.

4.3 Ablaufüberblick

Alice und Bob haben bisher nur unverschlüsselt über XMPP kommuniziert. Alice, mit JID *alice@server-x.de/pc*, weiß, dass Bob jetzt OMEMO benutzt und sie möchte eine verschlüsselte Unterhaltung starten. Bob ist gerade offline und Alice weiß nicht, auf welcher Ressource er zuerst verfügbar sein wird, weshalb sie die Nachricht an seine bloße JID, *bob@server-z.com*, verschickt. Alice hat auch eine XMPP-App auf ihrem Handy mit JID *alice@server-x.de/handy*. Folgendes muss geschehen, damit beide mit OMEMO-E2E-Verschlüsselung kommunizieren können:

1. Verifizierung von Bobs Geräten.
2. Verschlüsselung der ersten Nachricht an Bob (durch Alice's Client)
3. Entschlüsselung der ersten Nachricht an Bob (durch Bobs Client)
4. Ver-/Entschlüsselung der nachfolgenden Nachrichten

4.4 Verifizierung der öffentlichen Identitätsschlüssel

4.4.1 Auf OMEMO-Ebene

Alice kennt Bobs Geräteliste, da sie seinen PEP-Knoten abonniert hat. Sie kennt auch ihre eigene Geräteliste. Für jedes Gerät in beiden Gerätelisten – ihr eigenes ausgenommen – lädt sie jeweils das entsprechende Bündel herunter. Sie muss dann überprüfen, dass die Geräte tatsächlich Bob bzw. ihr selbst gehören. Sie vergleicht dazu jeweils die öffentlichen Identitätsschlüssel der jeweiligen Bündel über einen sicheren Kanal mit Bob bzw. mit ihrem anderen Gerät. Dies muss außerhalb von OMEMO stattfinden, indem ein Fingerabdruck bzw. der Schlüssel selbst verglichen wird (siehe Abbildung 4). Da Identitätsschlüsselpaare langfristiger Natur sind, braucht sie dies nur einmal pro Gerät zu tun. Nachdem die öffentlichen Identitätsschlüssel verifiziert sind, kann Alice mit dem nächsten OMEMO-Schritt fortfahren.

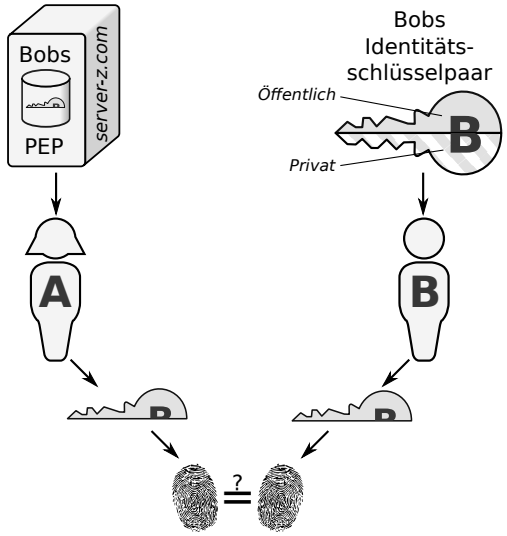


Abbildung 4 – Verifizierung von Bobs öffentlichem Identitätsschlüssel

4.4.2 Kryptografischer Hintergrund

Das Identitätsschlüsselpaar – sowie alle Schlüsselpaare von nun an – besteht aus Elliptische-Kurven-Schlüsseln [16]. Sowohl Alice als auch Bob generieren es einmalig bei der Installation des Clients. Jedes Gerät hat ein eigenes Identitätsschlüsselpaar.

Alice's Identitätsschlüsselpaar für das i-te Gerät $ID\_K^{A(i)}$ besteht also aus $ID\_K_{priv}^{A(i)}$ und $ID\_K_{pub}^{A(i)}$. Analog, besteht Bobs Identitätsschlüsselpaar für sein j-tes Gerät $ID\_K^{B(j)}$ aus $ID\_K_{priv}^{B(j)}$ und $ID\_K_{pub}^{B(j)}$.

4.5 Verschlüsselung der ersten ausgehenden Nachricht

4.5.1 Auf OMEMO-Ebene

Alice veranlasst eine symmetrische Verschlüsselung des Nachrichtentextes, der an Bob zu senden ist ① (siehe Abbildung 5). Daraus entsteht ein Geheimtext, den sie in ein OMEMO-Element (`<encrypted/>`) als `<payload/>` einpackt.

Anschließend nimmt sie für jedes Gerät in beiden Gerätelisten – ihr eigenes ausgenommen – den öffentlichen Identitätsschlüssel, den öffentlichen signierten Vorschlüssel sowie dessen Signatur aus dem Bündel und wählt ebenfalls daraus einen öffentlichen (unsignierten) Vorschlüssel zufällig aus ②. Mit diesen vier Komponenten baut sie eine Sitzung auf, sodass sie am Ende so viele Sitzungen wie Geräte aufgebaut hat ③. Mit jeder Sitzung kann sie an das entsprechende Gerät gerichtete Daten verschlüsseln.

Den Schlüssel, den sie für die symmetrische Verschlüsselung des Nachrichtentextes verwendet hat sowie zusätzliche Daten, die für die Entschlüsselung nötig sind, hängt Alice aneinander. Dann verschlüsselt sie für jedes Gerät das selbe Resultat mit den zuvor aufgebauten Sitzungen ④ und packt den resultierenden Geheimtext

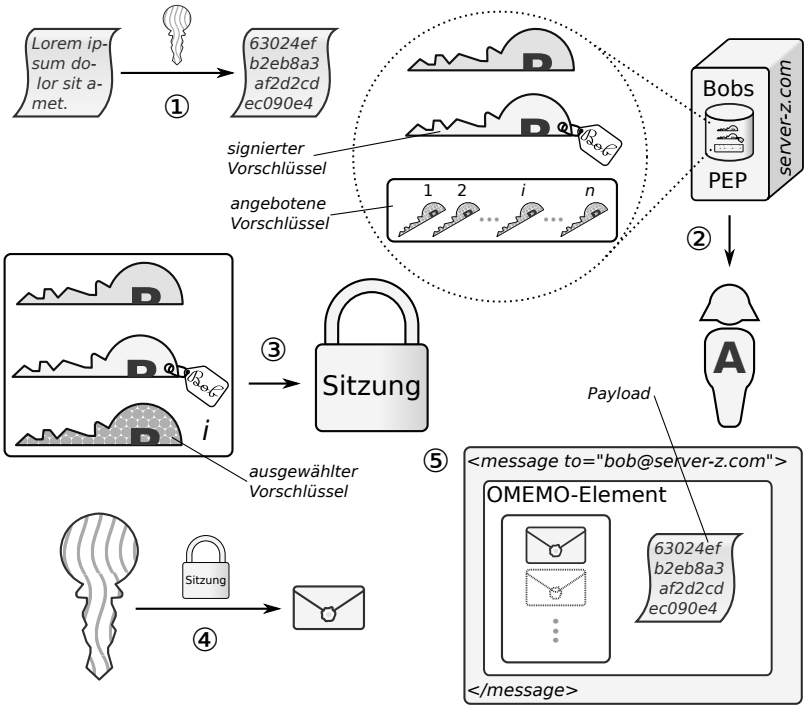


Abbildung 5 – Verschlüsselung der ersten ausgehenden Nachricht aus der Sicht der OMEMO-Spezifikation. Die Schritte ②, ③ und ④ werden pro zu empfangendes Gerät ausgeführt.

in ein OMEMO-Umschlag-Element (`<key/>`), das mit der ID des empfangenden Gerätes versehen wird.

Da dies die erste Nachricht an Bobs Geräte ist, sind die OMEMO-Umschlag-Elemente als Vorschlüssel-Umschläge gekennzeichnet. Alle Umschläge werden ins `<header/>` des OMEMO-Elements gepackt und letzteres mit der eigenen Gerät-ID als Absender versehen.

Das OMEMO-Element kann dann anstelle von `<body/>` in einer XMPP-Nachricht-Stanza an Bob verschickt werden ⑤. Unabhängig davon, auf welchem Gerät die Nachricht Bob erreicht, wird er sie entschlüsseln können, weil es ein Umschlag-Element für jedes Gerät gibt. Das Gleiche gilt für andere Geräte von Alice, die die Nachricht eventuell empfangen können (aus den in Abschnitt 3.1 erklärten Gründen).

4.5.2 Kryptografischer Hintergrund

Die Verschlüsselung des Nachrichteninhaltes erfolgt mit einem symmetrischen Verfahren (die Version 0.0.2 der Spezifikation schreibt AES GCM [17, 18] vor). Der dazu verwendete Schlüssel, sowie zusätzliche Daten, die für die Entschlüsselung nötig sind, werden konkateniert. Diese bilden den Klartext, der später mit den gebauten Sitzungen verschlüsselt und in die Umschläge gepackt wird.

Ein signierter Vorschlüssel im Bündel ist, so wie der öffentliche Identitätsschlüssel, der öffentliche Teil eines Schlüsselpaares. Jedes Gerät von Alice und Bob besitzt so ein Paar, das regelmäßig erneuert wird. Analog zum Identitätsschlüssel entspricht z. B. $SP\_K_{priv}^{A(i)}$ dem privaten signierten Vorschlüssel des i-ten Gerätes (d. h. des Gerätes mit ID *i*) von Alice.

Die Signatur im Bündel ($Sig_{pub}^{B(j)}$) ist das Ergebnis des Signierens von $SP\_K_{pub}^{B(j)}$ mit $ID\_K_{priv}^{B(j)}$.

Im Bündel befindet sich ebenfalls eine Menge von öffentlichen (unsignierten) Vorschlüsseln, aus der Alice einen zufällig auszuwählen hat ($P\_K_{pub}^{B(j)}$).

Jeder Vorschlüssel (signiert oder nicht) hat eine ID, sodass Alice Bob später mitteilen kann, welche seiner Schlüssel sie verwendet hat.

Der Sitzungsaufbau besteht aus zwei Schritten: der Vereinbarung eines Stammschlüssels und der Initialisierung eines Schlüsselableitungsmechanismus.

Vereinbarung eines Stammschlüssels Alice und Bob müssen nun einen Diffie-Hellman-Schlüsselaustausch [19] durchführen, d. h. sie müssen einen gemeinsamen Schlüssel vereinbaren, den nur sie kennen dürfen. Dafür verwenden sie das X3DH Schlüsselvereinbarungsprotokoll [20]. Vorteil des Protokolls ist, dass Bob offline sein darf, solange er im Voraus die nötigen öffentlichen Schlüssel und Signaturen veröffentlicht hat.

Zunächst überprüft Alice die Authentizität von $SP\_K_{pub}^{B(j)}$ anhand von $ID\_K_{pub}^{B(j)}$ und $Sig_{pub}^{B(j)}$. Wenn die Überprüfung erfolgreich ist, generiert sie ein kurzlebiges Schlüsselpaar $E\_K^{A(i)}$.

Danach hat sie alles, was nötig ist, um mit X3DH einen Schlüssel zu generieren, den auch Bob mit Hilfe von Alice's Bündel und $E\_K_{pub}^{A(i)}$ ableiten kann, wenn er online geht. Der neue gemeinsame Schlüssel $R\_k^{A(i)B(j)}$ ist die Ausgabe einer kryptografischen Hashfunktion angewendet auf die Konkatenation von vier X25519 Diffie-Hellman Berechnungen (DH-Berechnungen) [21], nämlich

$$DH(ID\_K_{priv}^{A(i)}, SP\_K_{pub}^{B(j)})$$
$$DH(E\_K_{priv}^{A(i)}, ID\_K_{pub}^{B(j)})$$
$$DH(E\_K_{priv}^{A(i)}, SP\_K_{pub}^{B(j)})$$
$$DH(E\_K_{priv}^{A(i)}, P\_K_{pub}^{B(j)})$$

in genau dieser Reihenfolge. Danach löscht Alice $E\_K_{priv}^{A(i)}$.

$R\_k^{A(i)B(j)}$ wird also aus häufig wechselnden Schlüsseln abgeleitet, die relativ oft vernichtet werden (siehe Abbildung 6), was *forward secrecy* gewährleistet, sowie aus verifizierten langlebigen Schlüsseln, die die gegenseitige Authentifizierung zwischen Alice und Bob möglich machen.

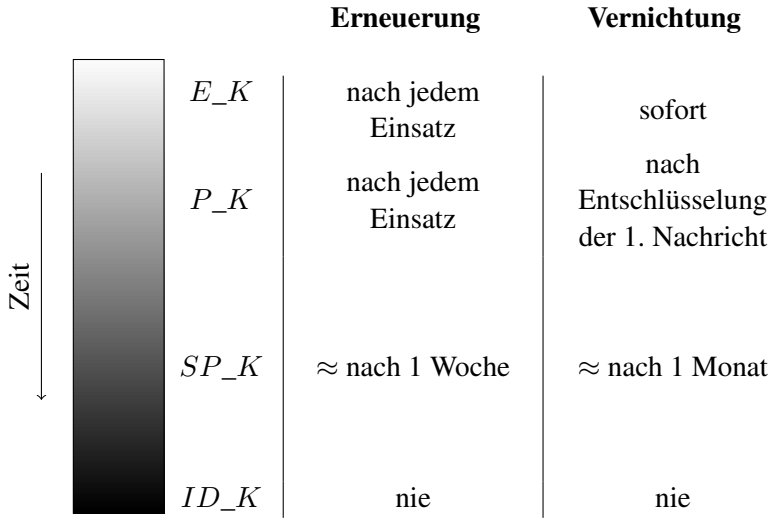


Abbildung 6 – Lebensdauer der Schlüsselpaare eines Gerätes

Initialisierung des Schlüsselableitungsmechanismus Alice verschlüsselt nichts direkt mit $R\_k^{A(i)B(j)}$, sondern initialisiert damit einen Mechanismus, der *für jede Nachricht* einen frischen Ver- bzw. Entschlüsselungsschlüssel generiert, den auch Bob ableiten kann, aber dessen Kompromittierung keine weiteren Schlüssel verrät. Dieser Mechanismus heißt *Double-Ratchet-Algorithmus* und wurde von T. Perrin und M. Marlinspike konzipiert (siehe Abbildung 7). Der Algorithmus wird ausführlich (jedoch sehr einfach) in [22] erklärt.

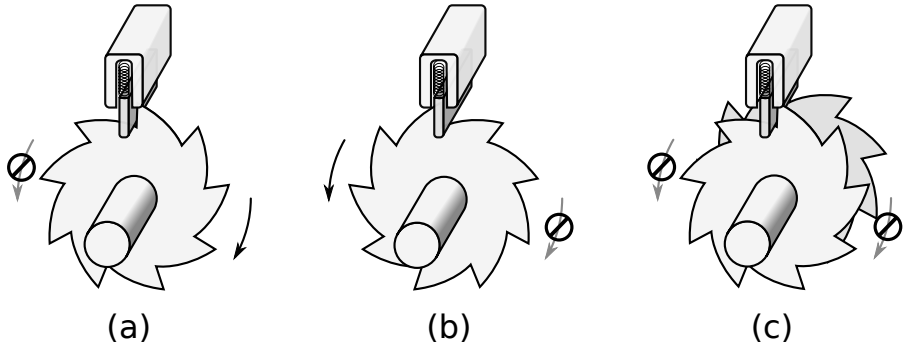


Abbildung 7 – Sperradmetapher (*ratchet*) im Falle der Kompromittierung eines Schlüssels k_i: (a) Mit k_i kann man k_{i+1} und damit k_{i+2} usw. ableiten, aber nicht k_{i-1} (die Achse dreht sich nur gegen den Uhrzeigersinn). (b) Mit k_i kann man k_{i-1} und damit k_{i-2} usw. ableiten, aber nicht k_{i+1} (die Achse dreht sich nur im Uhrzeigersinn). (c) *Nur* mit k_i kann man weder k_{i+1} noch k_{i-1} usw. ableiten (die Achse dreht sich nicht).

Die Initialisierung erfolgt, wenn Alice ein sog. Ratchet-Schlüsselpaar $RT\_K^{A(i)}$ generiert. Danach macht sie eine DH-Berechnung $DH(RT\_K_{priv}^{A(i)}, SP\_K_{pub}^{B(j)})$, die zusammen mit $R\_k^{A(i)B(j)}$ als Eingabe für den Anfang einer *Schlüsselsableitungskette* (SAK oder *KDF chain*) benutzt wird. Diese erste SAK wird *Stammkette* genannt. Aus den in der Stammkette abgeleiteten Schlüsseln werden zwei andere SAKs erzeugt, nämlich eine zur Ableitung von Verschlüsselungschlüsseln (*Versandkette*) und eine zur Ableitung von Entschlüsselungsschlüsseln (*Empfangskette*). Die so entstandene Versandkette wird zum Erzeugen von Verschlüsselungsschlüsseln benutzt, bis die erste Nachricht von Bob kommt.

Nachdem das Double-Ratchet initialisiert ist, hat Alice eine funktionierende Versandkette, mit der sie Vorschlüssel-Umschläge an Bob schicken kann (siehe Abbildung 8). In den Umschlag legt sie auch $ID\_K_{pub}^{A(i)}$, $E\_K_{pub}^{A(i)}$ und $RT\_K_{pub}^{A(i)}$. Bob baut seinen Teil der Sitzung erst auf, nachdem er den ersten Umschlag bekommen hat. Eine Sitzung ist nichts weiteres als der gespeicherte Zustand des Double-Ratchet.

4.6 Entschlüsselung der ersten eingehenden Nachricht

4.6.1 Auf OMEMO-Ebene

Bob empfängt die Nachricht auf seinem j-ten Gerät. Er muss dann ein Umschlag-Element im `<header/>` des OMEMO-Elements finden, das mit der ID seines Gerätes versehen ist. Da dies die erste Nachricht ist, die er von Alice bekommt, handelt es sich um einen Vorschlüssel-Umschlag.

Alice darf zu diesem Zeitpunkt offline sein, da der Umschlag alles beinhaltet, was für den Sitzungsaufbau nötig ist. Bob extrahiert die öffentlichen Schlüssel von Alice und die ID der verwendeten Vorschlüssel aus dem Umschlag, sodass er wissen

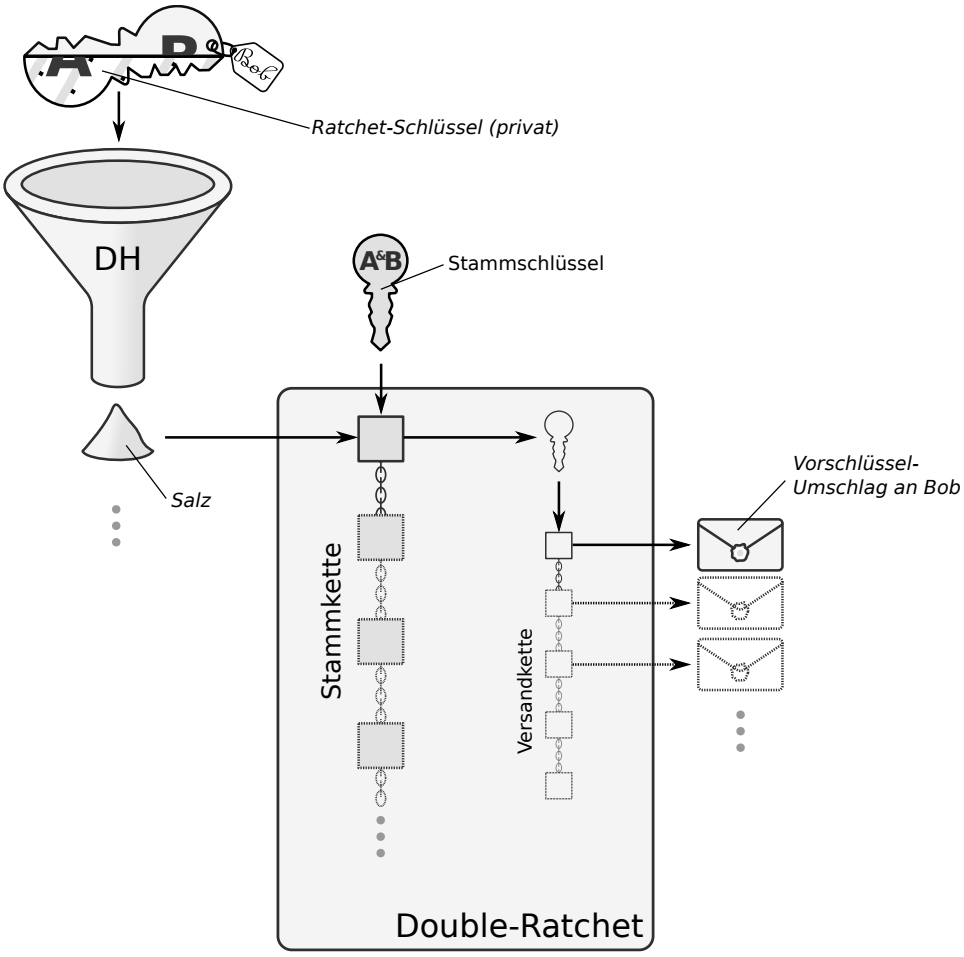


Abbildung 8 – Der Sitzungsaufbau ist die Initialisierung des Double-Ratchet. Alle Umschläge, die Alice verschickt, bevor sie die erste Antwort von Bob bekommt, heißen Vorschlüssel-Umschläge, weil sie mittels Bobs Vorschlüsseln generiert wurden, nämlich mit einem seiner unsignierten öffentlichen Vorschlüssel und mit seinem signierten öffentlichen Vorschlüssel, der auch als erster öffentliche Ratchet-Schlüssel benutzt wird.

kann, welche privaten Schlüssel er bei der Ableitung des gemeinsamen Schlüssels und bei dem Aufbau seines Teils der Sitzung benutzen soll.

Nachdem eine Sitzung mit Alice's Gerät auch seitens Bob besteht, kann er sie verwenden, um den Inhalt des Umschlages zu entschlüsseln. Das Ergebnis dieser Entschlüsselung ist wiederum der Schlüssel, sowie die zusätzlichen Daten, die Bob für die Entschlüsselung des Nachrichtentextes benötigt. Damit kann er zuletzt den im `<payload>`-Element gepackten Geheimtext in Klartext umwandeln.

Sollte die selbe Nachricht von anderen Geräten Bobs empfangen werden, sind sie auch in der Lage, den Inhalt zu entschlüsseln, solange es einen passenden Umschlag gibt.

4.6.2 Kryptografischer Hintergrund

Ableitung des Stammschlüssels Um $R\_k^{A(i)B(j)}$ abzuleiten, wendet Bob die gleiche kryptografische Hashfunktion wie Alice auf die Konkatenation von vier DH-Berechnungen, nämlich

$$DH(ID\_K_{pub}^{A(i)}, SP\_K_{priv}^{B(j)})$$
$$DH(E\_K_{pub}^{A(i)}, ID\_K_{priv}^{B(j)})$$
$$DH(E\_K_{pub}^{A(i)}, SP\_K_{priv}^{B(j)})$$
$$DH(E\_K_{pub}^{A(i)}, P\_K_{priv}^{B(j)})$$

in genau dieser Reihenfolge an. Sowohl $ID\_K_{pub}^{A(i)}$ als auch $E\_K_{pub}^{A(i)}$ hat er aus dem Umschlag extrahiert.

Initialisierung des Schlüsselableitungsmechanismus Analog zu Alice initialisiert Bob seine Stammkette mit $R\_k^{A(i)B(j)}$ und $DH(RT\_K_{pub}^{A(i)}, SP\_K_{priv}^{B(j)})$. Mit der Stammkette erzeugt er zunächst eine Empfangskette, die der Versandkette von Alice entspricht, sodass er die gleichen Schlüssel und in der selben Reihenfolge wie sie ableiten kann. Die so entstandene Empfangskette wird zum Erzeugen von Entschlüsselungsschlüsseln benutzt, bis der erste Nachrichtenaustausch vollständig ist, d. h. Bob hat seine erste Nachricht an Alice verschickt, sie hat sie empfangen und mit einer weiteren Nachricht geantwortet (siehe Abschnitt 4.7.2).

4.7 Ver-/Entschlüsselung von nachfolgenden Nachrichten

4.7.1 Auf OMEMO-Ebene

Nach der ersten Nachricht werden die Nachrichtentexte weiterhin mit neuen zufälligen Schlüsseln symmetrisch verschlüsselt, und letztere – nebst zusätzlicher Daten, die für die Entschlüsselung nötig sind – für jedes potenziell empfangenden Gerät in je ein `<key>`-Element gepackt, dessen Inhalt mit der bestehenden Sitzung erzeugt wird.

Diese Art von `<key>`-Elementen unterscheidet sich von der, die für die Initiierung des OMEMO-Nachrichtenverkehrs seitens Alice benutzt wurde. Insbesondere beinhaltet `<key/>` keine Informationen über Vorschlüssel, auch sind diese nicht nötig für dessen Verarbeitung. Deshalb werden sie einfach OMEMO-Umschlag genannt, im Gegensatz zu OMEMO-Vorschlüssel-Umschlag.

Die Empfangsprozedur verläuft analog: Passenden Umschlag finden, mit der bestehenden Sitzung entschlüsseln und mit dem Ergebnis den sich im `<payload/>` befindlichen tatsächlichen Inhalt der Nachricht in Klartext umwandeln.

4.7.2 Kryptografischer Hintergrund

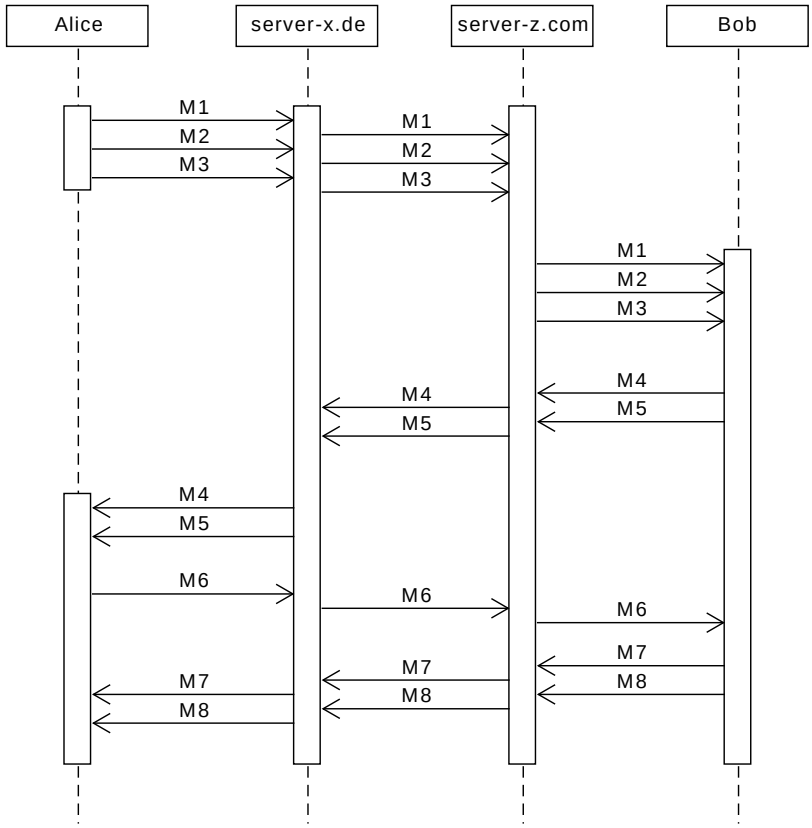
Wenn Bob die erste Antwort an Alice verschicken will, generiert er ein Ratchet-Schlüsselpaar $RT\_K^{B(j)}$ und berechnet $DH(RT\_K_{pub}^{A(i)}, RT\_K_{priv}^{B(j)})$. Damit füttert er die Stammkette zum Erzeugen einer Versandkette. $RT\_K_{priv}^{B(j)}$ löscht er unmittelbar danach.

Mit der Versandkette kann er dann Verschlüsselungsschlüssel ableiten, bis die nächste Nachricht von Alice kommt. In die Umschläge braucht er nun lediglich einen Schlüssel zu legen und zwar $RT\_K_{pub}^{B(j)}$. Dazu noch Sequenznummern, die Alice ermöglichen, den Umschlag eindeutig einer Position in ihrer Empfangskette zuzuordnen, falls die Nachrichten nicht in der ursprünglichen Reihenfolge ankommen.

Wenn die erste Nachricht von Bob ankommt, füttert Alice die Stammkette mit $DH(RT\_K_{priv}^{A(i)}, RT\_K_{pub}^{B(j)})$ ($RT\_K_{pub}^{B(j)}$ hat Bob an die Nachricht angehängt), leitet einen Schlüssel ab und startet damit eine (neue) Empfangskette, mit der sie dann die Nachricht entschlüsseln kann. Unmittelbar danach löscht sie $R\_k^{A(i)B(j)}$ und das alte Ratchet-Schlüsselpaar $RT\_K^{A(i)}$, generiert ein neues $RT\_K'^{A(i)}$ und leitet wieder einen Schlüssel aus der Stammkette ab, diesmal gefüttert mit $DH(RT\_K_{priv}'^{A(i)}, RT\_K_{pub}^{B(j)})$, womit sie eine neue Versandkette startet.

Anders als die Stammkette, die DH-Berechnungen als zusätzliche Eingabe für die Schlüsselableitung bekommt, haben die anderen SAKs keine externe Entropiezufuhr. Dafür sind sie aber kurzlebig, d. h. immer wenn Alice eine Nachricht von Bob empfängt, werden neue Schlüssel mit der Stammkette abgeleitet, die für neue Versand- und Empfangsketten benutzt werden. Wie oben genannt, wird eine bestimmte Versandkette zur Generierung von Verschlüsselungsschlüsseln benutzt, solange Alice Nachrichten an Bob verschickt, ohne dass er zwischenzeitlich geantwortet hat (siehe Abbildung 9). Die Empfangskette verhält sich analog.

Wenn Bob die zweite Nachrichtenreihe von Alice bekommt, ist der erste Nachrichtenaustausch vollständig. Er erneuert seine Versand- und Empfangsketten in einer analogen Weise, wie Alice im vorigen Schritt es getan hat. Alte Schlüssel werden gelöscht und so weiter.



Versand-/Empfangskette

⋮	⋮
$k-1$	$M1$, $M2$, $M3$
k	$M4$, $M5$, $M6$
$k+1$	$M7$, $M8$
⋮	⋮

Abbildung 9 – Beispiel für die Erneuerung von Nachrichtenableitungsketten im Nachrichtenverkehr. Jede Reihe von ankommenden Nachrichten treibt die Stammkette des Empfängers voran und somit löst sie eine Erneuerung der anderen SAKs aus. Wenn Bob Nachrichten $M1$, $M2$ und $M3$ von Alice bekommt, benutzt er zwar seine aktuelle Empfangskette ($k-1$), aber erzeugt danach neue Ketten in beide Richtungen, sodass $M4$ und $M5$ mit der neuen Versandkette (k) verschlüsselt werden. Beim Empfang erneuert Alice ihre SAKs auch und entschlüsselt $M4$ und $M5$ mit der neuen Empfangskette (auch k). $M6$ verschlüsselt sie ebenfalls mit ihrer k-ten Versandkette. Die SAKs werden so weiter erneuert ($M7$, $M8$, usw.).
Bob ist am Anfang offline. Alice zwischendurch.

5 Prototypentwicklung

Um die OMEMO-Spezifikation testen zu können, haben wir sie als Plugin für die Sofortnachrichtenbibliothek *libpurple* implementiert. Das Plugin ist freie Software und unter der *GNU Public License* zur Verfügung gestellt. Der Quellcode des Prototyps sowie seine Dokumentation können von `https://git.imp.fu-berlin.de/mancho/libpurple-omemo-plugin` heruntergeladen werden.

5.1 Verwendete Software

Im Folgenden werden die in der Implementierung eingesetzten Bibliotheken eingeführt, um die Erklärung der Entwurfsentscheidungen in Abschnitt 5.2 zu erleichtern.

5.1.1 *libpurple*

libpurple ist eine GPL-lizenzierte C-Bibliothek zur Unterstützung von Sofortnachrichtendiensten mit mehreren Protokollen [23]. Sie ist modular in Form von Plugins aufgebaut. Jedes zu unterstützende Protokoll wird als Plugin für den Bibliothekskern implementiert. Darunter befindet sich auch XMPP, das in Form eines sog. *Jabber-Plug-ins* zur Verfügung steht.

Für Drittentwickler bietet *libpurple* eine Plugin-API an, die das Laden von Plugins als dynamische Bibliotheken ermöglicht. Darüber hinaus stellt *libpurple* zusätzliche APIs sowie Signale bereit, die die Handhabung von Konten oder Verbindungsstreams bis hin zur XML-Verarbeitung erleichtern.

libpurple stammt aus dem Sofortnachrichtenclient Pidgin. *Pidgin* ist ein Mehrprotokollclient für Sofortnachrichtendienste. *libpurple* wurde von der Pidgin-Codebasis abgekoppelt und als unabhängiges Back-End strukturiert, sodass Pidgin schließlich nur ein GTK+-Front-End für *libpurple* ist. Derweil gibt es mehrere Front-Ends wie Finch (CLI) oder Adium (macOS). Allerdings orientiert sich *libpurple* besonders an Pidgin und wird mit ihm gemeinsam entwickelt.

Pidgin und *libpurple* unterscheiden vier Sorten von Plugins:

- *Core-Plugins* sind reine *libpurple* Plugins. Sie verwenden nur die von *libpurple* bereitgestellten APIs und haben keine Benutzeroberfläche. Damit können sie unabhängig vom Front-End eingesetzt werden. Das OMEMO-Plugin ist ein Core-Plugin.

- *Protokoll-Plugins* sind Core-Plugins, die *libpurple* um weiteren Sofortnachrichtenprotokolle erweitern. Das Jabber-Plugin ist ein Protokoll-Plugin.

- *UI-Plugins* sind Plugins, die eine Benutzeroberfläche anbieten, wodurch sie sich nur auf spezifische Front-Ends beschränken, z. B. Pidgin-Plugins, d. h. Plugins, die die grafische Schnittstelle von Pidgin modifizieren.

- *Loader-Plugins* sind eine spezielle Art von Plugins, die andere nicht in C geschriebene Plugin-Implementierungen laden können.

5.1.2 Signal Protocol Library

Als Implementierung des X3DH Schlüsselvereinbarungsprotokolls und des Double-Ratchet-Algorithmus wird die *Signal-Protocol-C-Bibliothek* eingesetzt (kurz Signal-Bibliothek). Die Signal-Bibliothek wurde von Open Whisper Systems entwickelt und unter der GPL-Softwarelizenz veröffentlicht. Open Whisper Systems erstellt Software für Sofortnachrichtendienste und Internettelefonie und betreibt sein eigenes Netzwerk namens Signal mit Clients für Android und iOS (*Signal-App*). Die C-Bibliothek ist eine Portierung der ursprünglichen Signal-Protocol-Java-Bibliothek in die Programmiersprache C. Es gibt auch eine Portierung auf Javascript.

Wir haben die Signal-Bibliothek gewählt, weil sie wie *libpurple* in C geschrieben ist, eine kompatible Softwarelizenz hat (GPLv3) und ihre kryptografischen Eigenschaften unabhängig geprüft sind [24, 25, 26].

Da die Signal-Bibliothek als Backend für die Signal-App und nicht für OMEMO konzipiert wurde, weichen manche Aspekte von der bisher eingeführten OMEMO-Logik ab, sodass beide Modelle in Übereinstimmung gebracht werden müssen.

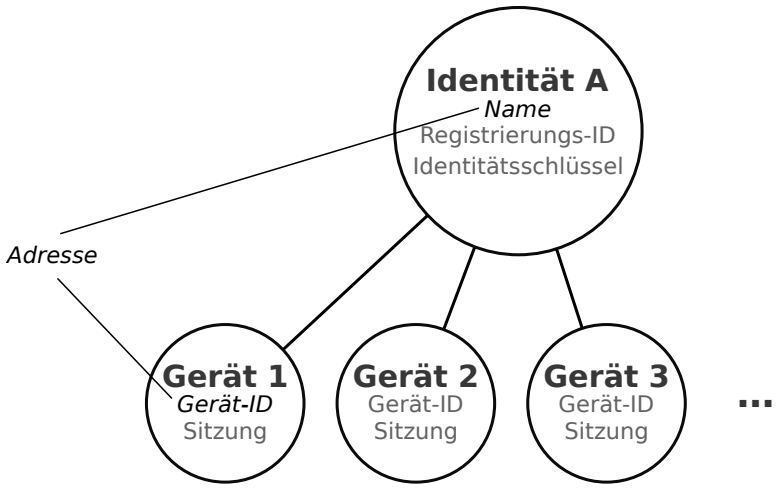


Abbildung 10 – Identitäts- und Geräte-Struktur, die sich aus der Schnittstelle der Signal-Bibliothek ableiten lässt

Geräte und Kontakte werden in der Signal-App anders als in OMEMO behandelt. Das Identitätsschlüsselpaar wird dem Kontakt zugewiesen, sodass alle Geräte eines Kontaktes den selben Identitätsschlüssel teilen (siehe Abbildung 10). Sowohl Kontakte als auch Geräte werden mit IDs versehen. Diese heißen jeweils Registrierungs- und Gerät-ID. Als zusätzliche Bezeichnung eines Kontakts wird ein Name benutzt, der der Telefonnummer in Signal entspricht. Der größte Nachteil des Signal-Ansatzes ist, dass der private Teil des Identitätsschlüssels an neue Geräte *out of band* übertragen werden muss, was mit dem OMEMO-Ansatz nicht nötig ist.

Um beide Modelle kompatibel zu machen und somit den Einsatz der Signal-Bibliothek zu ermöglichen, haben wir OMEMO-Geräte als einzelne Signal-Kontakte (sog. Identität) mit jeweils nur einem Signal-Gerät realisiert. Daraus entstehen die

Äquivalenzen von Tabelle 2. Die Konzepte dahinter sind gleich.

OMEMO	Signal
Kontakt	Identität
JID	Name
Gerät-ID	Registrierungs-ID
Identitätsschlüssel	Identitätsschlüssel
Sitzung	Sitzung
(JID, Gerät-ID)	Adresse

Tabelle 2 – Begriffsäquivalenzen zwischen OMEMO und Signal

Zur Datenserialisierung verwendet die Bibliothek Protocol Buffers. Sowohl die Sitzungen als auch die Schlüsselpaare werden damit in BLOBs umgewandelt, bevor sie persistent abgespeichert werden. Die dafür nötigen Routinen sind direkt in der Bibliothek enthalten.

Die primitiven kryptografischen Funktionen implementiert die Signal-Bibliothek nicht selbst. Sie stellt stattdessen eine Schnittstelle namens *Crypto-Provider* bereit, die an eine beliebige kryptografische Bibliothek angebunden werden kann, sofern diese einen sicheren Zufallsgenerator anbietet sowie Routinen zur Berechnung von Keyed-Hash Message Authentication Codes [27] (HMAC-SHA-256), SHA-512-Hashwerten und zur symmetrischen Ver- und Entschlüsselung mittels AES in verschiedenen Modi.

Zur Sicherung des Programmzustandes im sekundären Speicher stellt die Signal-Bibliothek ebenfalls eine Schnittstelle bereit. Sie besteht aus vier sog. *Stores* für jeweils Sitzungen, Vorschlüssel, signierte Vorschlüssel und Identitätsschlüssel. Anwender müssen diese Schnittstelle implementieren, damit der Verschlüsselungszustand zwischen Programmabläufen korrekt gespeichert und wiederhergestellt werden kann. Dabei ist der verwendete Speichermechanismus gleichgültig.

5.1.3 Weitere Software-Bibliotheken

Die Test-Routinen der Signal-C-Bibliothek verwenden *libcrypto* aus OpenSSL als Crypto-Provider. Aufgrund von Lizenzinkompatibilitäten und als *Proof of Concept* haben wir stattdessen *libgcrypt* von GnuPG verwendet. Dank der modularen Art des Crypto-Providers ist aber ein Wechsel auf ein anderes kryptografisches Back-End leicht realisierbar.

Die Stores sind in Form einer SQLite3-Datenbank realisiert, die je aktives XMPP-Konto angelegt wird, d. h. Konto A (*alice@server-x.de*) erhält seine eigene Datenbank, sowie Konto B (*alice@server-y.net*) usw.. Jede Datenbank bedient ein einzelnes Store-Kontext-Objekt (mehr dazu in Abschnitt 5.2.3).

Zwecks Plattformunabhängigkeit werden soweit möglich Abstraktionen aus der GLib-Bibliothek benutzt anstelle reiner C-Konstruktionen. Die GLib-Bibliothek wird bereits umfangreich von *libpurple* eingesetzt, weshalb dies der natürliche Ansatz im OMEMO-Plugin ist.

Für die Verarbeitung von XML verwendet das Plugin die XML-API von *libpurple* (`xmlnode.h`). Allerdings musste diese ergänzt werden, da zwei für OMEMO erforderliche Operationen fehlten: eine zum Entfernen des Textinhalts eines Elementes und eine zum Entfernen eines Unterbaums in einem Dokument. Diese wurden implementiert und zum Plugin-Code hinzugefügt.

5.2 Entwurf

Nächstfolgend werden die wichtigsten Entwurfsentscheidungen erläutert, die bei der Entwicklung des Prototyps getroffen wurden.

5.2.1 Datenrepräsentation

Beim Entwurf des Plugins haben wir festgestellt, dass die Daten, mit denen unser Programm arbeiten würde, von zwei Quellen stammen bzw. an zwei Stellen verschickt würden. Diese sind in Abbildung 11 dargestellt. Auf der einen Seite befindet sich der persistente Lokalspeicher in Form der lokalen Datenbank, in der der Programmstatus zwischen den Ausführungen serialisiert gesichert und aus der der Programmstatus wiederhergestellt wird. Auf der anderen Seite befindet sich das XMPP-Netzwerk. Dies schließt u. a. PEP-Veröffentlichungen, PEP-Benachrichtigungen, Nachrichtenversand und Nachrichtenempfang ein.

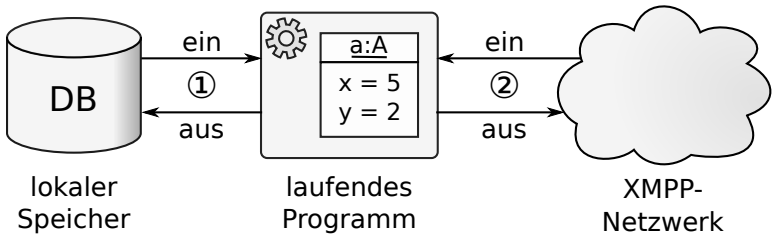


Abbildung 11 – Datenaustausch des laufenden Programms

Das Datenformat beider Stellen ist unterschiedlich. Für die Sicherung in der lokalen Datenbank müssen die Daten meistens in binäre Form serialisiert bzw. aus dieser Form deserialisiert werden ①. Die Daten, die über den XMPP-Stream empfangen werden, sind als strukturierter Text dargestellt (XML) ebenso wie die Daten, die auch darüber ins Netz verschickt werden ②. Somit gibt es drei mögliche Darstellungen der selben Daten: zur Aufbewahrung, für die Programmlogik und zur Netzübertragung.

Die Datenrepräsentation zur Netzübertragung wurde in Abschnitt 4.2 erläutert. In Abbildung 12 wird das vorgeschlagene Datenbankschema dargestellt, das zur Aufbewahrung auf Sekundärspeicher benutzt wird. Diese Darstellung wurde vom OMEMO-XML-Namensraum sowie von der Signal-Bibliothek und den Anforderungen der Programmlogik beeinflusst.

Die Information über die entfernten Geräte wird in den Tabellen *contacts* und *devices* gespeichert. Für jeden Kontakt wird die bloße JID gespeichert sowie ein

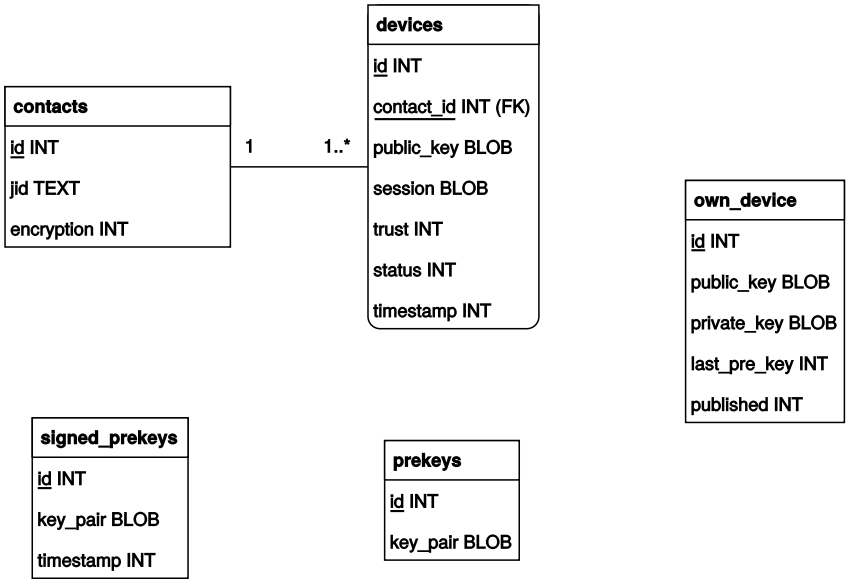


Abbildung 12 – Schema der persistenten Datenbank für den lokalen Speicher

Flag, der bestimmt, ob die OMEMO-E2E-Verschlüsselung zwischen dem lokalen Gerät und diesem Kontakt aktiviert ist. Dies kann der Endbenutzer bestimmen. Je Kontakt gibt es mindestens ein Gerät. Die Eindeutigkeit der Gerät-IDs wird dadurch erzwungen, dass der Primärschlüssel der Tabelle *devices* aus Gerät-ID *und* Kontakt besteht, und somit indirekt aus Gerät-Tupeln. Solange eine Sitzung mit einem Gerät besteht, wird sie im binären Format der Signal-Bibliothek serialisiert als *session* gespeichert. Sollte ein entferntes Gerät aus der PEP-Geräteliste des Kontakts gelöscht worden sein, wird es mit dem *status* `INACTIVE` versehen, damit keine Umschläge mehr an dieses Gerät versendet werden. Das *timestamp* Feld beinhaltet die Zeit der letzten Statusänderung, sodass alte Geräte eventuell gelöscht werden können. Darüber hinaus enthält *devices* das Feld *trust*, um die Vertrauensverwaltung zu realisieren (siehe Abschnitt 5.3.5).

Die Komponenten des Bündels des lokalen Clients werden in den Tabellen *own_device*, *signed_prekeys* und *pre_keys* gespeichert. Jeder Eintrag eines signierten Vorschlüssels wird mit seiner Erzeugungszeit gekennzeichnet (*timestamp*), um die rechtzeitige Erneuerung und Löschung zu ermöglichen. Da unsignierte Vorschlüssel gleich nach dem Einsatz verworfen werden, ist diese Kennzeichnung in *prekeys* nicht notwendig. Die Schlüsselpaare sind im binären Format der Signal-Bibliothek serialisiert. Die Tabelle *own_device* beinhaltet nur einen Eintrag mit dem privaten und öffentlichen Teil des Identitätsschlüsselpaares sowie einen Zeiger zur Behandlung der Vorschlüssel-IDs als Indizes eines Ringpuffers (*last_pre_key*) und einen Flag, der gesetzt wird, sobald die ID des lokalen Gerätes zum ersten Mal über PEP in die entfernte Geräteliste eingetragen wird. Letzterer als Teil des Algorithmus, der die Eindeutigkeit der eigenen Gerät-ID erzwingt (siehe Abschnitt 5.3.1).

Um die Programmlogik möglichst unabhängig von den Darstellungen zu ma-

chen, haben wir versucht, folgende Fragen zu beantworten: Welche Daten werden ausgetauscht bzw. sind im Programm notwendig und wie kann man sie optimal strukturieren? Daraus entstanden fünf abstrakte Datentypen: drei sind die gleichen wie im OMEMO-Namensraum, nämlich „Geräteliste“, „Bündel“ und „OMEMO-Element“. Die übrigen sind „Umschlag“ und „Gerät“. Für diese Datentypen haben wir Operationen bereitgestellt, die die Darstellungsumwandlung übernehmen. Sollte sich das XML- oder Datenbankschema ändern, ist damit nur eine Modifizierung der entsprechenden Operation notwendig, ohne die Logik des Hauptprogramms bearbeiten zu müssen. Ein Überblick dieser Datentypen wird in Abbildung 13 dargestellt.

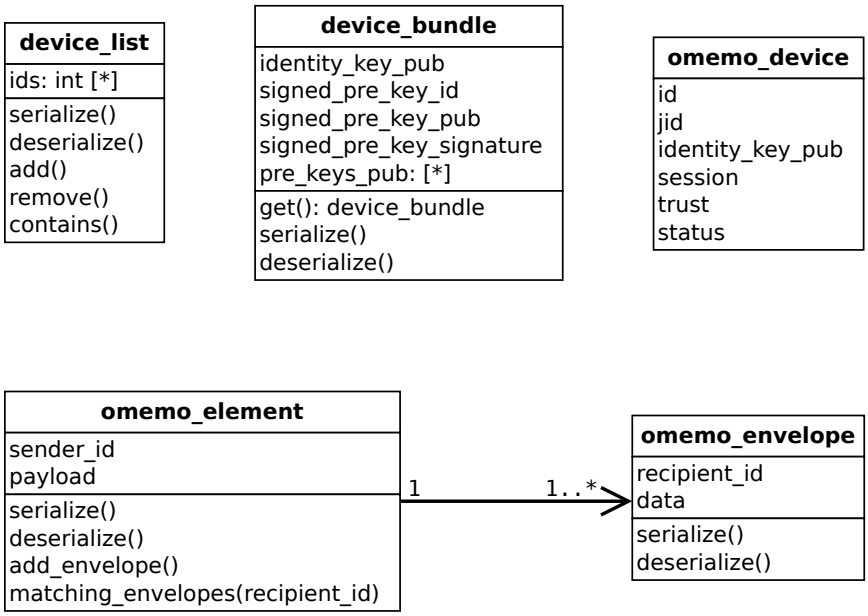


Abbildung 13 – Abstrakte Datentypen

Außer *omemo_device* stellen sämtliche Typen mindestens zwei Operationen zur Verfügung: `serialize()` und `deserialize()`, die dem Darstellungswechsel von und in XML entsprechen. *device_bundle* besitzt außerdem eine Operation, die die Daten, die in der Datenbank gespeichert sind, deserialisiert. Diese wird `get()` statt `deserialize()` genannt, um sie von den XML (De)Serialisierungs-Operationen zu unterscheiden.

Eine Geräteliste wird durch *device_list* abstrahiert, unabhängig davon, ob es sich um die eigene oder die eines Kontaktes handelt, da die Struktur in beiden Fällen gleich ist. Sie hat die kanonische Schnittstelle einer Menge. Somit werden Duplikate vermieden und Suchoptimierungen ermöglicht.

Das Bündel des lokalen Gerätes wird durch *device_bundle* abstrahiert. Die Attribute werden anhand der Datentypen dargestellt, die die Signal-Bibliothek anbietet. Die `get`-Operation, die die in der Datenbank gespeicherten Daten aufruft und deserialisiert, agiert in diesem Sinne als Konstruktor.

OMEMO-Elemente werden häufig verarbeitet. Sei es in ausgehenden oder in ankommenden Nachrichten. Sie werden durch *omemo_element* abstrahiert und beinhalten mindestens einen OMEMO-Umschlag, der seinerseits durch *omemo_envelope* abstrahiert wird. Bei den Serialisierungsoperationen ruft eine Instanz von *omemo_element* die entsprechende Operation von *omemo_envelope* auf. Der Inhalt des Umschlags wird als binäre Daten in *data* gespeichert, im Format der Signal-Bibliothek. Ebenso binär ist der Geheimtext der Nachricht in *payload* gespeichert. *sender_id* ist die ID des Absender-Geräts und *recipient_id* die eines potentiellen Empfänger-Geräts. Die Operation `matching_envelopes()` ist beim Empfang eines OMEMO-Elementes notwendig, da es mehrere Umschläge mit der selben Gerät-ID geben kann (siehe Abschnitt 5.3.4).

Zuletzt werden Geräte durch *omemo_device* abstrahiert. Dieser Datentyp dient der Kommunikation mit der Datenbank und entspricht einem Eintrag in der Tabelle *devices*. Da ein Gerät, wie in der Datenbank konzipiert, nicht für die Übertragung an andere geeignet ist, verzichten wir auf Operationen zur Umwandlung in/von XML. Die Struktur ist nur für die einfache Manipulation von Geräte-Informationen innerhalb des Plugins gedacht.

5.2.2 Integration mit dem XMPP-Plugin

Die Plugin-API von *libpurple* ermöglicht nur die Implementierung von allgemeinen Anwendungsfällen auf protokollübergreifender Ebene. Da das OMEMO-Plugin protokollspezifisch ist, ist diese Funktionalität unzureichend – z. B. ist es mit der Plugin-API allein nicht möglich, ein- und ausgehende XMPP-Stanzas abzufangen.

Das Jabber-Plugin bietet zwar eine Ereignis-API an, die das Abfangen von Stanzas erlaubt, aber weitere Routinen, die das OMEMO-Plugin voraussetzt, sind nicht aus externen Plugins sichtbar – es gibt z. B. keine öffentliche API-Routine, die dem Plugin erlaubt, Gerätelistenbenachrichtigungen über PEP zu abonnieren, obwohl das Jabber-Plugin dies intern implementiert.

Nach Rücksprache mit den Entwicklern von *libpurple* ist folgende Lösung gefunden: Die sonst privaten Header-Dateien des Jabber-Plugins werden in das OMEMO-Plugin eingebunden, sodass die Programmsymbole des ohnehin geladenen Plugins (`libjabber.so`) aus OMEMO sichtbar sind. Da das OMEMO-Plugin das Jabber-Plugin voraussetzt, kann es nicht dazu kommen, dass dieses in einer Instanz von *libpurple* geladen wird, die nicht über das Jabber-Plugin verfügt. Der Nachteil dieses Ansatzes ist, dass eine Modifikation der Programmierschnittstelle des Jabber-Plugins entsprechende Anpassungen im OMEMO-Plugin vonnöten macht. Dies haben wir jedoch als Kompromiss zwischen Modularität und Komplementarität betrachtet und sind diesen eingegangen. Für den Fall, dass sich die Programmierschnittstelle des Jabber-Plugins ändert, haben wir detailliert dokumentiert, welche Teile des Jabber-Plugins im OMEMO-Plugin benutzt werden, damit der Vergleich der Änderungsprotokolle in Zukunft einfacher ablaufen kann. Die Integration aller bisher genannten Komponenten ins *libpurple* Framework wird in Abbildung 14 dargestellt.

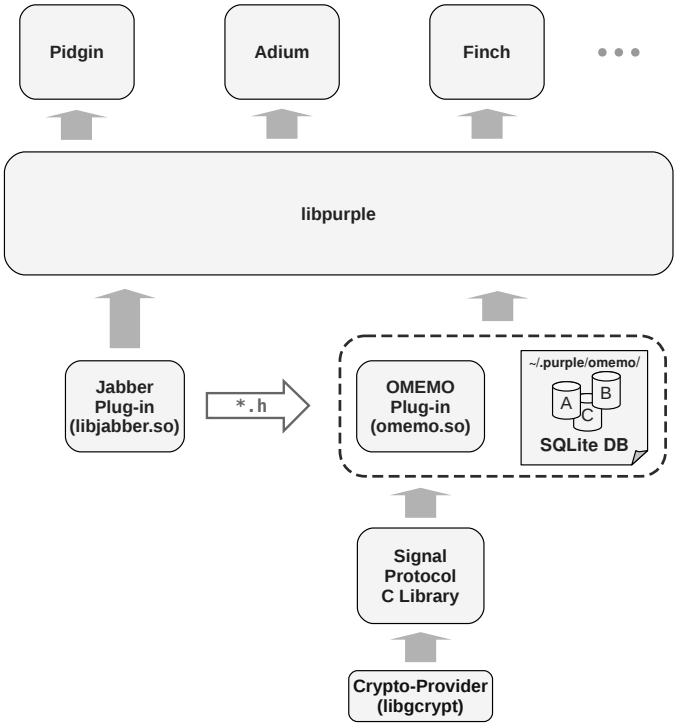


Abbildung 14 – OMEMO-Plugin (gestrichelt) als Teil des *libpurple* Frameworks

5.2.3 Kontext

libpurple unterstützt nicht nur mehrere Protokolle, sondern auch die gleichzeitige Verwaltung von mehreren Konten, sodass die meisten Routinen auf dem Client im Kontext eines spezifischen Kontos auszuführen sind. Das OMEMO-Plugin ist ebenfalls so entworfen, d. h. es werden so viele Programmzustände gesichert, wie XMPP Konten auf dem Client vorhanden sind. Deshalb muss das Plugin Kontextobjekte in einer ähnlichen Weise wie *libpurple* zur Verfügung stellen und sie seinen eigenen Routinen übergeben. Dazu wird von der Konto-API Gebrauch gemacht, die ein solches Objekt vom Typ *PurpleAccount* anbietet.

Die meisten Routinen des Plugins sind konto-spezifisch und benötigen mindestens den Zugriff auf den lokalen Speicher, d. h. auf die Datenbank des entsprechenden Kontos. Routinen, die direkt die Signal-Bibliothek verwenden, bekommen einen Datenbank-Handle (*Database*), aus dem sich ein *signal_protocol_store_context*-Objekt konstruieren lässt, das die verschiedenen Stores des aktuellen Kontos der Signal-Bibliothek zur Verfügung stellt. Darüber hinaus gibt es Routinen, die auf Ereignisse aus der *libpurple*-API reagieren oder Stanzas verschicken. Diese bekommen entweder *PurpleAccount* oder *PurpleConnection* als Kontextobjekte und können dadurch Subroutinen aufrufen, da sie in der Lage sind, Datenbank-Objekte und Store-Kontexte zu konstruieren. Rückruffunktionen (*callbacks*) können auch *JabberStream*-Strukturen als Kontext übergeben bekommen. Aus Abbildung 15 geht hervor, dass

dies ebenfalls den Zugriff zu den anderen Kontextobjekten gewährleistet.

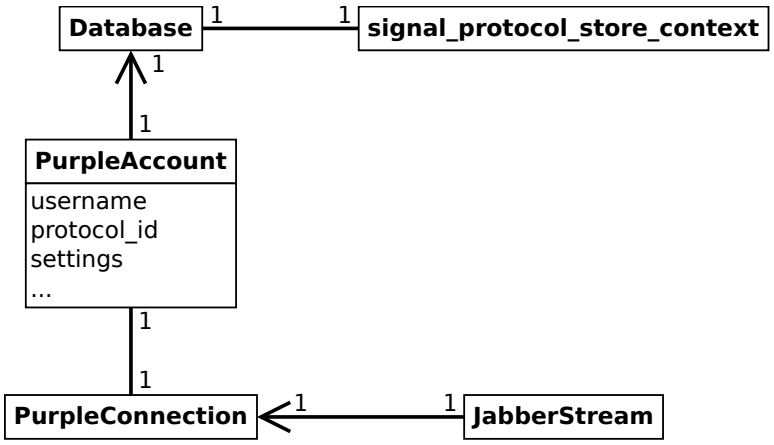


Abbildung 15 – Kontextobjekte für die Plugin-Routinen

5.3 Umsetzung

Das OMEMO-Plugin wird in Form einer dynamischen Bibliothek (`omemo.so`) in *libpurple* geladen. Beim Laden wird überprüft, dass das Jabber-Plugin ebenfalls vorhanden ist. Für jedes aktive XMPP-Konto wird eine OMEMO-Initialisierung durchgeführt, falls nötig. Anschließend werden die benötigten Rückruffunktionen mit den passenden Signalen verbunden und der Ereignishandler für OMEMO-PEP-Benachrichtigungen beim Jabber-Plugin registriert.

Wir beschränken uns hier auf die Kernaspekte der Plugin-Implementierung. Für eine detaillierte Einsicht siehe das Git-*Repository* des Projektes.

5.3.1 Initialisierung der OMEMO-Komponenten

Die OMEMO-Initialisierung ist die Erstellung und Speicherung der Daten, die ein XMPP-Konto in *libpurple* für die verschlüsselte Kommunikation mittels OMEMO braucht. Dies findet in `omemo_account_setup()` statt und kann in drei Schritte geteilt werden: Generierung der Signal-Elemente, Generierung des lokalen Speichers und Veröffentlichung des Schlüsselmaterials.

Generierung der Signal-Elemente Hier werden direkt die von der Signal-Bibliothek zur Verfügung gestellten Funktionen vom *signal_protocol_key_helper* angewendet. Es wird die Registrierungs-ID, das Identitätsschlüsselpaar und ein signierter Vorschlüsselpaar mit zufälliger ID erzeugt. Außerdem werden so viele Vorschlüsselpaare generiert, wie in `OPTIMAL_PRE_KEY_COUNT` konfiguriert ist. Die Anfangs-ID für die Vorschlüssel wird ebenfalls zufällig gewählt. `OPTIMAL_PRE_KEY_COUNT` ist ein C-Makro, das die optimale Anzahl von Vorschlüsseln beinhaltet,

die das Plugin zu jeder Zeit versucht, anderen Geräten bereitzuhalten (mehr dazu in Abschnitt 5.3.5).

Generierung des lokalen Speichers Die SQLite3-Datenbank, die als lokaler Speicher fungiert, muss an das entsprechende *libpurple*-Konto gebunden sein (siehe Abbildung 15). Dazu wird eine zufällige alphanumerische Zeichenkette generiert, die dann anhand der Konto-API von *libpurple* als Konto-Einstellungsparameter *omemo-db-id* gespeichert wird. Die Datenbankdatei wird danach ins Konfigurationsverzeichnis des Nutzers angelegt, welches in Unix-Systemen üblicherweise `$HOME/.purple/` ist. In diesem Verzeichnis legt das Plugin ein weiteres für OMEMO dediziertes Verzeichnis an, um die Batenbankdateien von allen konfiguruerten Konten zu speichern. Wenn die *omemo-db-id* eines Kontos *24qm2x8s1u92ozga* ist, ist die entsprechende Datenbankdatei `$HOME/.purple/omemo/24qm2x8s1u92ozga.sqlite`. Nach Erstellen der Datenbank wird sie mit den Signal-Elementen befüllt.

Immer beim Laden des Plugins wird überprüft, ob *omemo-db-id* in den Einstellungen gespeichert ist und ob die Datenbankdatei existiert. Sollte dies nicht der Fall sein, wird eine erneute Installation durchgeführt. Darüber hinaus wird der Inhalt der Datenbank nach fälligen Aktualisierungen kontrolliert, d. h. wenn das Datenbankschema veraltet ist, wird eine Migrationsroutine ausgelöst. Dies basiert auf dem SQLite3 spezifischem Befehl `PRAGMA` [28, S. 381]. Das Plugin speichert die aktuelle Version des Datenbankschemas im *user-version*-Pragma und die Migrationsroutine kann davon abhängige inkrementelle Änderungen anwenden, bis das Schema der laufenden Plugin-Version entspricht.

Veröffentlichung des Schlüsselmaterials Zuletzt wird aus den generierten Signal-Elementen das Gerät-Bündel konstruiert und veröffentlicht, d. h. als neuer PEP-Knoten hochgeladen. Außerdem wird das neu entstandene Gerät zur PEP-Geräteliste des XMPP-Kontos hinzugefügt. Allerdings schreibt das OMEMO-XEP vor, dass der Client die Geräteliste nach Vorhandensein der frisch generierten Gerät-ID vor der ersten Veröffentlichung durchsuchen und, falls nötig, eine neue generieren muss, um ihre Eindeutigkeit zu garantieren. Da der Aufruf der Geräteliste mittels IQ-Stanzas erfolgt, ist `publish_device()` eine asynchrone Routine. Dies stellt ein Problem dar, weil ID-Duplikate erst nach der Ausführung von `omemo_account_setup()` erkannt werden. Außerdem gibt es keine Garantie dafür, dass das Konto zur Zeit der Installation von OMEMO online ist. Ist dies nicht der Fall, muss die Veröffentlichung bis zur nächstmöglichen Verbindung hinausgeschoben werden.

Aufgrund dieser Beeinträchtigungen war es notwendig, einen Flag in die Datenbank einzutragen, der die allererste Veröffentlichung des Gerätes signalisiert, sodass dies als Sonderfall von `publish_device()` behandelt und eine neue Installation ausgelöst werden kann, wenn ein ID-Duplikat festgestellt wird. In Auflistung 2 heißt dieser Flag *first_publishing* und hat vor der Installation den Wert *true*, da das

Gerät unveröffentlicht ist. Erst nachdem ID-Duplikate ausgeschlossen sind, wird der Flag abgeschaltet (Zeile 13). Es ist zu bemerken, dass die Ausführung ab Zeile 4 erst nach der PEP-Antwort fortgesetzt wird, sodass die Zeilen 4-13 in der Praxis als Teil einer Rückruffunktion ausgeführt werden.

Auflistung 2 – Pseudocode von `publish_device()`

```
1  publish_device(my_id)
2  {
3      devlist = fetch_device_list() // asynchrone PEP Abfrage
4      if (my_id in devlist) {
5          if (first_publishing) {
6              reinstall
7              publish_device()
8          }
9      }
10     else {
11         add my_id to devlist
12         push_device_list(devlist) // PEP Upload
13         first_publishing = false
14     }
15 }
```

5.3.2 Synchronisation

Beim Starten des Clients und bevor die in *libpurple* konfigurierten Konten online gehen, erfolgt das Abonnement der PEP-Knoten der Gerätelisten. Der entsprechende Knoten-Namensraum wird der Liste der Fähigkeiten (*capabilities*), die der Client unterstützt, hinzugefügt. Danach erhält der Client PEP-Benachrichtigungen nach der in Abschnitt 3.3 erklärten Logik. Der Client bekommt Benachrichtigungen nicht nur von anderen Kontakten, sondern auch von der eigenen JID. Jede eingetroffene Geräteliste muss mit der lokalen Datenbank abgeglichen werden. Dies findet in `device_list_update_cb()` wie folgt statt:

- Wenn ein Gerät mit `status=ACTIVE` lokal, aber nicht entfernt, vorhanden ist, wird es deaktiviert, d. h. es wird nicht mehr für dieses Gerät bei ausgehenden Nachrichten mitverschlüsselt.
- Wenn ein Gerät mit `status=INACTIVE` lokal *und* entfernt vorhanden ist, wird es wieder aktiviert, d. h. es handelt sich um ein wieder erscheinendes Gerät.
- Wenn ein Gerät entfernt, aber nicht lokal, vorhanden ist, wird ein neues Gerät-Tupel mit `status=ACTIVE` in die Datenbank eingetragen, d. h. es handelt sich um ein neues Gerät.

Sollte die Geräteliste von der eigenen JID stammen, d. h. von der JID des lokalen *libpurple*-Kontos, müssen zusätzliche Operationen durchgeführt werden:

- Das eigene Bündel wird veröffentlicht. Dies ist zwar nicht notwendig, da das Schlüsselmaterial sich nicht geändert hat, und führt zum erhöhten Netzwerkverkehr-Overhead, aber macht die Implementierung robuster gegenüber Fehlverhalten von anderen Clients, insbesondere wenn Dritte lokale Versionen der Geräteliste nach einem PEP-Ausfall wiederherstellen. Ohne diese Maßnahme würde es in solchen Fällen im PEP-Speicher eine Referenz auf ein nicht existierendes Bündel geben, die der betroffene Client nicht anhand der Geräteliste allein feststellen kann. Dies würde die Verfügbarkeit des Clients beeinträchtigen, da kein Bündel aufrufbar wäre.
- Wenn die eigene Gerät-ID nicht vorhanden ist, wird sie zur entfernten Geräteliste hinzugefügt. Außer nach einem PEP-Ausfall kann dies auch dann passieren, wenn zwei Geräte beinahe gleichzeitig sich selbst zum ersten Mal auf der Geräteliste bekanntgeben, ohne von dem anderen zu wissen. Da das Hinzufügen zur Geräteliste tatsächlich als das Überschreiben des PEP-Knotens implementiert ist, würde das zweite Gerät das erste überschreiben. Mit dieser vom XEP vorgeschriebenen Überprüfung bei jeder eingetroffenen (eigenen) Geräteliste ist diese Wettlaufsituation behoben.

Immer wenn ein Gerät-Tupel lokal aktiviert wird, wird in der Datenbank die entsprechende Zeitmarke (`timestamp`-Feld) aktualisiert. Sie ist als „zuletzt gesehen“ zu interpretieren und dafür gedacht, regelmäßiges Aufräumen zu ermöglichen. Die Aktualisierung der Zeitmarke wird direkt auf Datenbankebene anhand von SQL-Triggers [28, S. 109] implementiert. Außer in den o. g. Situationen werden die Gerät-Tupel ebenfalls aktiviert, wenn Nachrichten vom entsprechenden Gerät empfangen werden.

Darüber hinaus wird das Gerät-Bündel immer veröffentlicht – d. h. der PEP-Knoten wird überschrieben –, wenn Vorschlüssel gebraucht werden, also nach der Entschlüsselung von Vorschlüssel-Umschlägen und wenn das lokale Konto online geht.

5.3.3 Nachrichtenversand

Wenn OMEMO-Verschlüsselung für einen Kontakt aktiviert ist, werden Klartextnachrichten, d. h. Nachricht-Stanzas mit `<body/>`, vom Plugin abgefangen und an `encrypt_message()` weitergegeben, sodass sie am Ende in Form von Nachricht-Stanzas mit einem OMEMO-Element (`<encrypted/>`) und ohne `<body/>` versendet werden. Es gibt jedoch kein Signal in der Jabber-API von *libpurple*, das ausschließlich ausgehende Nachricht-Stanzas abfangen lässt, sodass das OMEMO-Plugin alle XMPP-Stanzas anhand vom API-Signal *jabber-sending-xmlnode* abfangen und daraus die Nachricht-Stanzas absondern muss.

Zunächst wird die Routine `prepare_encryption()` aufgerufen, die die Vertrauenswürdigkeit und die Sitzungen der empfangenden Geräte überprüft. Sollte eine Sitzung fehlen, wird einen neue aufgebaut (in `build_missing_sessions()`). Das entsprechende Gerät-Bündel wird über PEP abgefragt und daraus wird ein zufälliger Vorschlüssel ausgewählt. Anhand dieses Vorschlüssels, des signierten Vorschlüssels, der Signatur, des Identitätsschlüssels und der OMEMO-Gerät-ID (als Registrierungs-ID) wird ein *session_pre_key_bundle* in der Signal-Bibliothek erzeugt, das an eine signaleigene Sitzungsaufbauroutine weitergegeben wird. Das *session_pre_key_bundle* enthält zusätzlich eine Gerät-ID, die in OMEMO von keiner Bedeutung ist (Abschnitt 5.1.2). Ihr Wert wird also auf 0 gesetzt.

Danach wird der Inhalt vom ursprünglichen `<body/>` wie in Abschnitt 4.5.1 beschrieben verschlüsselt. Anschließend wird ein *omemo_element* erstellt und der resultierende Geheimtext als *omemo_element→payload* gesetzt. Danach werden die Umschläge generiert und zu *omemo_element* hinzugefügt: ein *omemo_envelope* je empfangendes Gerät. Der Klartext-Inhalt sämtlicher Umschläge ist immer der gleiche, nämlich das Geheimnis zur Entschlüsselung von *omemo_element→payload* und wird in Geheimtextform als *omemo_envelope→data* gespeichert. In diesem Schritt werden als empfangende Geräte diejenigen betrachtet, die folgende Bedingungen erfüllen:

(a) Geräte des Nachrichtenempfängers, die aktiv und vertrauenswürdig sind und mit denen eine Sitzung besteht.

(b) Weitere Geräte des Nachrichtensenders, d. h. die dem lokalen Gerät nicht entsprechen, die aktiv und vertrauenswürdig sind und mit denen eine Sitzung besteht.

Das nun vollständige *omemo_element* wird dann als XML serialisiert und anstelle von `<body/>` zur Nachricht-Stanza hinzugefügt. Ebenfalls zur Nachricht-Stanza hinzugefügt wird ein Behandlungshinweis (aus XEP-0334: *Message Processing Hints* [29]), damit sie serverseitig wie eine Nachricht-Stanza mit `<body/>` behandelt wird. Obwohl die OMEMO-Spezifikation dies nicht vorschreibt, wird die Stanza zusätzlich mit einem `<encryption>`-Element aus XEP-0380: *Explicit Message Encryption* [30] gekennzeichnet, sodass Clients, die OMEMO-E2E-Verschlüsselung nicht unterstützen, die Nachricht korrekt behandeln bzw. ignorieren.

5.3.4 Nachrichtenempfang

Eingehende Nachricht-Stanzas, die ein OMEMO-Element enthalten, werden vom Plugin abgefangen und an `decrypt_message()` weitergegeben, sodass sie am Ende in Form von Nachricht-Stanzas mit dem entschlüsselten Klartext im `<body/>` dem Client präsentiert werden.

Im Gegensatz zur Situation beim Versenden, gibt es in der Jabber-API ein Signal *jabber-receiving-message*, das ausschließlich eingehende Nachricht-Stanzas abfangen lässt, damit das OMEMO-Plugin nicht alle Stanzas überprüfen muss.

Jedoch gibt es keine Namensraum-Registrierung, die nur Nachricht-Stanzas mit OMEMO-Elementen beobachten lässt. In jedem `<message/>` muss deshalb das Plugin nach einem OMEMO-Element suchen und, falls gefunden, die Verarbeitung übernehmen.

Zunächst wird das OMEMO-Element aus XML deserialisiert. Das daraus resultierende *omemo_element* enthält dann einen oder mehrere Umschläge (*omemo_envelope*), innerhalb derer einer zu finden ist, dessen *recipient_id* mit der eigenen Gerät-ID übereinstimmt.

Handelt es sich um einen Vorschlüssel-Umschlag, wird er an die Routine `decrypt_pre_key_message()` weitergeleitet. Dort wird der Inhalt des Umschlages mithilfe der Signal-Routine `session_cipher_decrypt_pre_key_message()` entschlüsselt und das zugehörige Gerät-Tupel in der DB mit *status* `ACTIVE` markiert. Diese Routine baut gleichzeitig eine neue Sitzung mit dem sendenden Gerät auf, die eine eventuell vorhandene alte Sitzung überschreibt.

Handelt es sich um einen sonstigen Umschlag, bedeutet dies, dass eine Sitzung bereits besteht. Der Umschlag wird dann an die Routine `decrypt_signal_message()` weitergeleitet. Dort wird der Inhalt des Umschlags mit einer anderen Signal-Routine entschlüsselt, nämlich mit `session_cipher_decrypt_signal_message()`. Diese Routine baut keine Sitzung auf, sondern benutzt die bestehende Sitzung, die aus vorherigem Nachrichtenaustausch stammt. Wie im Fall eines Vorschlüssel-Umschlags wird das Gerät-Tupel als aktiv markiert.

Sollte bei der Entschlüsselung des Umschlags ein Vorschlüssel verbraucht worden sein, muss er vom eigenen PEP-Bündel gelöscht und durch einen frischen ersetzt werden. Dazu generiert das Plugin neue Vorschlüssel (mehr Details dazu in Abschnitt 5.3.5) und ruft danach `publish_bundle()` auf.

Sollte der Vorschlüssel-Umschlag einen bereits von einem anderem Gerät benutzten Vorschlüssel verwenden, scheitert der Sitzungsaufbau, da der Vorschlüssel schon vernichtet ist. In diesem Fall wird gemäß Spezifikation eine Sitzung seitens des Empfängers (des lokalen Geräts) aufgebaut und ein Vorschlüssel-Umschlag mit zufälligem Inhalt in einem sonst leeren OMEMO-Element (einem sog. *Ratchet Update Message*) dem sendenden Gerät in einer Nachricht-Stanza zugeschickt.

Ist der Umschlag erfolgreich entschlüsselt und *omemo_element→payload* ist nicht leer, wird `process_message_element()` aufgerufen. Diese Funktion nimmt den Klartext des Umschlags und verwendet ihn als Entschlüsselungsschlüssel für *omemo_element→payload* . Daraus entsteht der Klartext der empfangenen Nachricht, welcher an die eingehende Nachricht-Stanza als Inhalt von `<body/>` angehängt wird.

Die Prozedur zum Empfangen einer Nachricht muss dennoch ein potenzielles Problem bekämpfen, das in Version 0.2 des OMEMO-XEP unerwähnt bleibt. Wie in Abschnitt 4.1 erwähnt, muss die ID eines Gerätes eindeutig unter den Geräten eines Kontakts sein, nicht aber bezüglich anderer. Infolgedessen kann es Situationen geben, in denen ein OMEMO-Element zwei Umschläge mit der gleichen *recipient_id* enthält. Dies würde dazu führen, dass nur eines der empfangenden Geräte

das Element erfolgreich entschlüsseln kann, nämlich dasjenige, dessen Umschlag zuerst im OMEMO-Element vorkommt. Das andere Gerät, das fälschlicherweise den selben Umschlag zu entschlüsseln versucht, wird die Nachricht verwerfen müssen. Ob ein Gerät die Nachricht in so einer Situation entschlüsseln kann, hängt davon ab, welcher Umschlag im OMEMO-Element als erster vorkommt bzw. wie die Umschlagsliste durchsucht wird.

Als Lösung wurde `decrypt_message()` so implementiert, dass die Suche nach einem passenden Umschlag die Möglichkeit von *recipient_id*-Duplikaten berücksichtigt. Dazu werden zuerst alle übereinstimmenden Umschläge mittels `matching_envelopes()` aus dem OMEMO-Element extrahiert und über diese Menge die Entschlüsselung versucht, nicht nur über die erste Übereinstimmung.

5.3.5 Vertrauens- und Schlüsselverwaltung

Wenn die erste Nachricht eines bestimmten Gerätes eintrifft und einen mit dem lokalen Gerät übereinstimmenden Vorschlüssel-Umschlag enthält, wird daraus der öffentlicher Teil des Identitätsschlüsselpaares extrahiert und in der Datenbank als *devices.public_key* gespeichert. Dies umgeht absichtlich die Store-Routine `save_identity()`, die sonst in der Signal-Bibliothek diese Aufgabe erledigt, weil diese nach der Identitätsstruktur der Signal-App arbeitet, wo mehrere Geräte die selbe Identität teilen, weshalb sie keine Gerät-ID als Parameter akzeptiert.

Jeder Identitätsschlüssel wird somit genau einem Gerät-Tupel zugewiesen. Jedes Tupel hat eine gewisse Vertrauenswürdigkeit, die in direkter Verbindung mit der Verifizierung des Identitätsschlüssels über einen sicheren Kanal außerhalb von OMEMO steht. Das sog. *Vertrauen* ist in der Datenbank als *devices.trust* gespeichert und hat drei mögliche Werte:

- `TRUSTED` bedeutet, dass der öffentliche Identitätsschlüssel des Gerätes verifiziert ist. Der Endbenutzer hat explizit das Gerät als vertrauenswürdig eingestuft und es ist sicher, OMEMO-Nachrichten dahin zu verschicken.
- `UNTRUSTED` bedeutet, dass der öffentliche Identitätsschlüssel noch nicht verifiziert wurde oder dass der Identitätsschlüssel kompromittiert ist, weshalb kein an dieses Gerät gerichteter Inhalt verschlüsselt werden darf. Der Endbenutzer hat explizit das Gerät als nicht vertrauenswürdig eingestuft und es ist nicht sicher, OMEMO-Nachrichten dahin zu verschicken.
- `UNDECIDED` bedeutet, dass der Endbenutzer keine Vertrauenseinstufung für dieses Gerät durchgeführt hat. Dieser ist der Anfangswert jedes Tupels. Das Versenden von OMEMO-Nachrichten an das Gerät ist nicht erlaubt. Es werden jedoch eingehende Nachrichten entschlüsselt.

Die Vertrauensverwaltung des Prototyps richtet sich nach einem konservativen Ansatz: das Vertrauen *muss* explizit vom Endbenutzer gesetzt werden. Nur nach dem er die Schlüsselverifizierung durchgeführt und das Gerät entweder als vertraut

oder unvertraut gesetzt hat, wird es überhaupt zur Verschlüsselung berücksichtigt. Ferner wird eine Entscheidung gezwungen, indem gar keine ausgehende Kommunikation mit dem Kontakt möglich ist, solange mindestens ein Gerät unentschiedenes Vertrauen besitzt. Die Motive und Schwierigkeiten, die mit diesem Ansatz gebunden sind, werden in Abschnitt 5.4 erklärt.

Obwohl die Signal-Bibliothek eine Vertrauensüberprüfung eingebaut hat, orientiert sich diese an einem anderen Vertrauensmodell. Die Signal-Store-Routinen-Schnittstelle `is_trusted_identity()` unterstützt ein dichotomisches Vertrauensmodell nach dem *Trust-On-First-Use*-Prinzip (TOFU), womit Vertrauen nicht vom Endbenutzer gesetzt wird und ein Identitätsschlüssel nur dann als nicht vertrauenswürdig eingestuft ist, wenn dieser sich nachträglich ändert. Ist ein Gerät nicht vertrauenswürdig, baut die Signal-Bibliothek keine Sitzung auf. Im OMEMO-Plugin gibt es dagegen, wie bereits erläutert, drei mögliche Vertrauensstufen und obwohl keine Daten an nicht vertrauenswürdige Geräte verschickt werden, muss eine Sitzung zur Entschlüsselung von eingehenden Nachrichten existieren. Aufgrund dieser Inkompatibilität haben wir `is_trusted_identity()` so implementiert, dass sie alle Schlüssel als vertrauenswürdig betrachtet. Somit wird die Vertrauensüberprüfung der Signal-Bibliothek *de facto* deaktiviert und vom Plugin übernommen. Diese Vorgehensweise ist auch im OMEMO-XEP empfohlen.

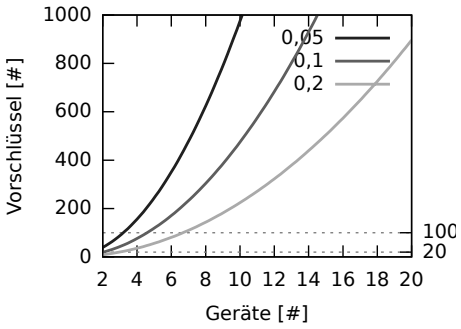
Die Wartung von (signierten) Vorschlüsseln findet vor jeder Veröffentlichung des Gerät-Bündels statt. Die `publish_bundle()`-Routine überprüft, ob der Vorschlüsselspeicher nachgefüllt werden muss, ob eine Erneuerung des aktuellen signierten Vorschlüssels fällig ist und ob alte signierte Vorschlüssel vernichtet werden müssen. Entsprechende Subroutinen werden ggf. aufgerufen.

Eine Erneuerung des signierten Vorschlüssels ist notwendig, wenn *signed_prekeys.timestamp* mehr als `SIGNED_PRE_KEY_DAYS_TO_RENEW` in der Vergangenheit liegt. Ein alter signierter Vorschlüssel wird vernichtet, wenn mehr als `SIGNED_PRE_KEY_DAYS_TO_DELETE` Tage seit *signed_prekeys.timestamp* vergangen sind. In der aktuellen Implementierung beträgt die Zeit zur Erneuerung und Vernichtung jeweils 7 und 30 Tage.

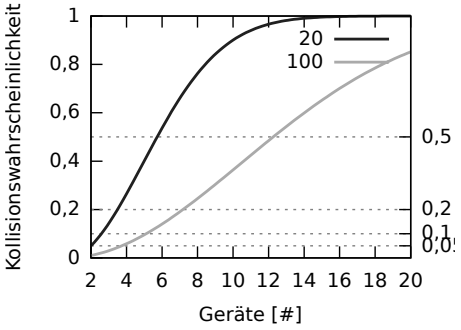
Wie in Abschnitt 5.2.1 erwähnt, enthält *own_device.last_pre_key* die ID des zuletzt generierten Vorschlüssels. Der Vorschlüsselspeicher wird nachgefüllt, wenn die Anzahl der Vorschlüssel `OPTIMAL_PRE_KEY_COUNT` unterschreitet. In solchen Fällen werden so viele neue Vorschlüssel wie nötig generiert und mit IDs versehen, die mit $(\textit{own\_device.last\_pre\_key} + 1) \bmod (2^{16} - 1)$ anfangen.

Die Wahl eines Wertes von `OPTIMAL_PRE_KEY_COUNT` impliziert einen Kompromiss zwischen Netzwerkverkehr-Overhead und Kollisionswahrscheinlichkeit. Je niedriger der Wert, desto wahrscheinlicher sind Kollisionen beim zufälligen Auswählen des Vorschlüssels, wenn mehr als ein Gerät versucht, einen Vorschlüssel-Umschlag an das selbe Gerät zu verschicken, ohne dass letzteres zwischendurch ein neues Bündel hochgeladen hat. Je höher, desto größer wird das Bündel, das ins PEP-Speicher hochgeladen werden muss.

Um das Verhalten der Kollisionswahrscheinlichkeit p zu veranschaulichen, haben wir die Verhältnisse zwischen den relevanten Variablen anhand des Geburtstags-



(a) Anzahl de Vorschlüssel im Bündel, um Kollisionswahrscheinlichkeiten jeweils unter 5%, 10% und 20% zu erreichen.



(b) Obere Schranke der Kollisionswahrscheinlichkeit für Bündel mit jeweils 20 und 100 Vorschlüsseln.

Abbildung 16 – Analyse der Kollisionswahrscheinlichkeit p nach der assymptotischen Approximation $g \approx \sqrt{2k \cdot \ln\left(\frac{1}{1-p}\right)}$ von Ahmed und McIntosh [31], wobei g und k jeweils die Anzahl von Geräten und Vorschlüsseln sind.

problems modelliert. Sei g die Anzahl der Geräte, die gleichzeitig eine Vorschlüssel-Nachricht an das selbe Gerät verschicken, wobei „gleichzeitig“ bedeutet, dass keine Veröffentlichung des Bündels des Empfängers dazwischen stattfindet, z. B. weil das Gerät offline ist. Das Bündel enthält k Vorschlüssel, aus denen jedes von den g Geräten einen auswählen muss. Je nach Anwendungsfall gibt es gezielte Bündelgröße und maximale Toleranz von Kollisionswahrscheinlichkeit.

Pidgin ist (i) ein Desktop-Client (im Gegensatz zu mobilen Clients) und (ii) setzt XMPP als Sofortnachrichtendienst ein (im Gegensatz zu sonstigen Anwendungen von XMPP, wie z. B. IoT) . Aus diesen Tatsachen haben wir zwei Kriterien abgeleitet, die unserer Meinung nach eine akzeptable Leistung nachweisen und in direkter Verbindung mit den o. g. Variablen stehen:

(a) *Zielgröße:* Die Gesamtgröße der Vorschlüsseln bleibt unter 10 kB.

(b) *Zielwahrscheinlichkeit:* Die Kollisionswahrscheinlichkeit bleibt unter 10%, wenn gleichzeitig 4 Geräte eine Vorschlüssel-Nachricht dem selben Gerät senden.

Wegen (i) haben wir eine geringe Kollisionswahrscheinlichkeit einem geringen Netzwerkverkehr-Overhead vorgezogen. Wegen (ii) haben wir 4 Geräte als der plausibelste Extremfall genommen. Die Größe des Bündels hängt von der Anzahl der Vorschlüssel k ab. Ein üblicher Vorschlüssel serialisiert als `<pre-key-public/>` besteht aus ca. 90 Bytes Text, sodass die Größe des Elternelements `<pre-keys/>` ungefähr $0,09 \cdot k$ kB beträgt.

Der Kompromiss zwischen Zielgröße und Zielwahrscheinlichkeit wird in Abbildung 16 dargestellt. Für eine bestimmte Zielwahrscheinlichkeit muss die Größe

des Bündels immer wieder erhöht werden, je mehr Geräte gleichzeitig Nachrichten verschicken (16a). Für eine gegebene Zielgröße ist die obere Schranke der Kollisionswahrscheinlichkeit bereits bei wenigen Geräten, die gleichzeitig Nachrichten verschicken, sehr hoch (16b), wie aus dem Geburtstagsproblem bekannt ist. Um die zwei o. g. Kriterien zu erreichen, haben wir uns in der Implementierung des Plugins für ein `OPTIMAL_PRE_KEY_COUNT` von 100 entschieden, das über dem im XEP empfohlenen Minimalwert von 20 liegt. Damit beträgt die Größe von `<pre-keys/>` insgesamt ca. 9 kB und iegt die Kollisionswahrscheinlichkeit unter 6%. Im Fall von 2 Geräten ist sie sogar unter 1%.

5.4 Verbesserungsmöglichkeiten

Als Core-Plugin hat die Implementierung nur eingeschränkten Zugriff auf die GUI-Elemente des Clients. Vielmehr besitzen manche *libpurple*-Clients gar keine grafische Oberfläche. Deshalb ist keine grafische Vertrauens- und Schlüsselverwaltung im OMEMO-Plugin vorhanden. Um die Vertrauenswürdigkeit eines Schlüssels einzustufen, muss der Nutzer direkt in der Datenbank die Werte ändern. Der vorläufige Ansatz besteht darin, das Versenden einer Nachricht abzubrechen, sobald das Vertrauen in ein Gerät unentschieden ist, damit der sendende Nutzer dazu gezwungen ist, eine Vertrauensentscheidung zu treffen und diese in die Datenbank einzutragen. Dies ist selbstverständlich eine erhebliche Beeinträchtigung der Benutzbarkeit, die zu beheben ist. Außerdem gibt es keine sichtbaren Rückmeldungen im Falle von Fehlern oder Warnungen. Nur in der Debug-Konsole ist es möglich, zu sehen, wenn etwas fehlschlägt. Eine Lösung wäre, das OMEMO-Plugin für *libpurple* durch ein weiteres UI-Plugin zu ergänzen, das front-end-spezifisch gedacht ist – z. B. ein Pidgin-Plugin – und das eine GUI anbietet, die vor dem Senden den Nutzer zu einer Vertrauensentscheidung auffordert, sodass der Vorgang zwar unterbrochen, aber nicht abgebrochen werden muss. Diese Ergänzung gehört jedoch nicht zum Ziel dieser Arbeit.

Während die ganze Prozedur zum Versenden einer Nachricht korrekt aussieht, wenn man von synchronen Operationen ausgeht, könnte sie problembehaftet sein, wenn diese Annahme wegfällt. Wenn es in `prepare_encryption()` zu einem Sitzungsaufbau kommt, wird pro Sitzung das jeweilige Bündel über PEP abgefragt. Dies geschieht mittels einer IQ-Abfrage, d. h. der Sitzungsaufbau ist eine asynchrone Operation, sodass es möglich ist, dass bei der anschließenden Bestimmung der Menge empfangender Geräte manche ignoriert werden, weil für sie keine Sitzung rechtzeitig vorhanden ist, d. h. sie erfüllen die Bedingungen in Abschnitt 5.3.3 nicht. Dies ist besonders bei der allerersten OMEMO-Nachricht zwischen zwei Kontakten merkbar und die übliche Folge davon ist, dass das OMEMO-Element gar keine oder nur Umschläge für Geräte des sendenden Kontakts enthält. Im aktuellen Prototyp wird lediglich vor der Serialisierung von *omemo_element* überprüft, ob es überhaupt Empfängergeräte gibt, die dem Empfängerkontakt gehören. Anderenfalls wird der Versand abgebrochen. Das ist weder optimal, noch behebt es das Problem. Für den Nutzer äußert sich das als eine verschickte Nachricht, die am anderen

Ende verworfen wird. Ein Lösungsansatz wäre, eine Nachrichtenwarteschlange zu implementieren, in der Nachrichten, die noch nicht versendet werden können, vorübergehend aufbewahrt und erst nach dem Sitzungsaufbau versendet werden. Dazu aber müsste das OMEMO-Plugin in die Programmlogik des Nachrichtenversands von *libpurple* eingreifen und nicht mehr als ein einfacher Inhaltsmodifikator von ausgehenden Nachrichten agieren. Solch ein Ansatz birgt potenzielle Konflikte mit anderen Plugins und auch mit dem *libpurple*-Kern selbst. Eine weitere Alternative wäre, ähnlich wie die Lösung für die Vertrauensverwaltung, auf das Problem GUI-seitig einzugehen indem während des Sitzungsaufbaus eine Arbeitsflussunterbrechung erzwungen wird, sodass Nachrichtenverarbeitung oder das Versenden weiterer Nachrichten verhindert wird, bis die Sitzungen aufgebaut sind. Die Realisierbarkeit dieses Ansatzes hängt jedoch von den Berechtigungen eines UI-Plugins und seiner Fähigkeit, die Benutzerschnittstelle zu blockieren, ab.

Bisher haben wir *libgcrypt* als Crypto-Provider benutzt. Allerdings verwendet *libpurple* bereits Crypto-Primitive von GnuTLS, einer weiteren kryptografischen Bibliothek, die ebenfalls die Voraussetzungen erfüllt, um als Crypto-Provider zu fungieren. Ein Wechsel auf GnuTLS wäre deshalb nicht nur sinnvoll, um die Drittsoftwareabhängigkeiten des OMEMO-Plugin zu reduzieren, sondern auch aufgrund des modularen Aufbaus des Crypto-Providers leicht umsetzbar.

6 Leistungsauswertung des Prototyps

6.1 Versuchsaufbau

Um die Leistung unseres Plugins zu evaluieren, haben wir einen komparativen Test durchgeführt, indem ein simulierter Nachrichtenaustausch mit zufälligem aber fixem Inhalt jeweils in Klartext und OMEMO-verschlüsselt stattfindet.

Da ein Test mit zwei entfernten XMPP-Servern eine hohe Ergebnisabweichung aufgrund unkontrollierter Variaben im Netzwerkverkehr als Folge hätte, sollte der Nachrichtenweg möglichst kurz bleiben. Dazu haben wir zwei Kontakte, *alice@localhost* und *bob@localhost*, mit jeweils einem Gerät auf einem lokalen XMPP-Server eingerichtet. In dieser Weise findet keine Server-zu-Server-Kommunikation statt und die Netzwerk-Latenz bleibt minimal.

Bei jedem Testvorgang wurde die Zeit t zwischen dem Versand der ersten und dem Empfang der letzten Nachricht gemessen. Des Weiteren wurden andere Variablen wie Speichernutzung, CPU-Last und Laufwerksdurchsatz mitgeschnitten.

6.2 Testanwendung

Für die Durchführung des Tests haben wir eine kleine Anwendung in C geschrieben, die sich in *libpurple* als Plugin einbinden lässt und n Nachrichten abwechselnd von Alice an Bob und von Bob an Alice verschickt. Ob die Nachrichten als Klartext oder OMEMO-verschlüsselt verschickt werden, hängt davon ab, ob das OMEMO-Plugin zurzeit aktiviert ist. Der Inhalt jeder Nachricht ist eine zufällig generierte

Zeichenkette mit bis zu 140 Zeichen. Alle Testvorgänge wurden auf einer Maschine mit einem Intel® Core™ i3-3225 4-Kern-Prozessor, 8 GB DDR3-Speicher und Samsung 840 Pro SSD-Laufwerk durchgeführt. Als XMPP-Server wurde Prosody 0.9.10 und als Client Finch 2.10.12 (Kommandozeilenschnittstelle) eingesetzt.

6.3 Ergebnisse

In Abbildung 17 ist die durchschnittliche Dauer nach 6 Vorgangswiederholungen mit jeweils 10, 100, 1000 und 10000 Nachrichten mit und ohne Verschlüsselung dargestellt. Beim 6. Vorgang mit n=10000 war die marginale Variation der Durchschnittsdauer ca. ±0,003, weshalb wir auf weitere Wiederholungen verzichtet haben.

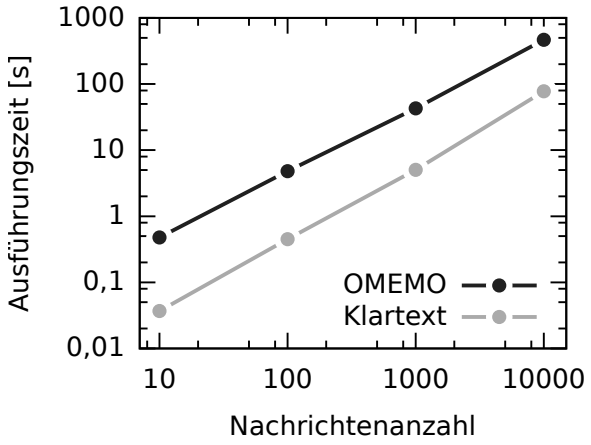


Abbildung 17 – Gemessene Ausführungszeiten (in Sekunden) bzgl. der Nachrichtenanzahl n

Wie erwartet ändert sich die Zeitkomplexität des Nachrichtenaustausches bezüglich n mit der Verschlüsselung im untersuchten Bereich nicht (die Messungen weisen auf $O(n)$ hin), jedoch unterscheiden sich die tatsächlichen Ausführungszeiten um mindestens eine Größenordnung. Dies ist zwar im durchschnittlichen Fall eines kleinen n für den Endbenutzer im Millisekunden-Bereich und somit kaum merkbar, aber bei einer größeren Anzahl an Nachrichten wird der Zeitunterschied immer ausgeprägter (z. B. bei n=1000 4 s im Klartext gegen 40 s mit OMEMO). Dies deutet auf ein Nachrichtendurchsatzproblem, d. h. auf einen Flaschenhals in der Bearbeitung von OMEMO-Nachrichten, hin. Ein Blick auf den Overhead bei CPU, Speicher und E/A liefert Anhaltspunkte für eine genauere Einschätzung des Ursprungs des Zeitunterschieds.

Wir haben einen Overhead-Indikator konstruiert, indem wir während jedes Vorgangs mit n=1000 im 100 ms-Takt die Speichernutzung in Kilobytes, die prozentuale CPU-Auslastung und den Datendurchsatz beim Schreiben auf sekundären Speicher in Kilobytes pro Sekunde aufgezeichnet haben. Wir haben dann den Durchschnittswert jeder Variablen pro Vorgang und wiederum den Durchschnitt aller Vorgänge

berechnet ($\bar{v_i}$ für die i-te Variable). Schließlich haben wir den Overheadfaktor der drei Variablen als $\frac{\bar{v_i}^{OMEMO} - \bar{v_i}^{Klartext}}{\bar{v_i}^{Klartext}}$ berechnet. Ein Wert von 0 bedeutet also keinen Overhead. Ein Wert von x bedeutet, dass mit OMEMO ein $x\bar{v_i}^{Klartext}$ höherer Wert gemessen wurde. In Abbildung 18 sind diese Messergebnisse angezeigt.

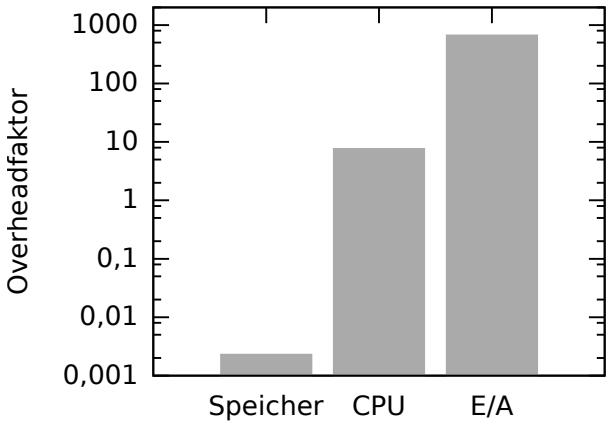


Abbildung 18 – Overheadfaktor $\left(\frac{\bar{v_i}^{OMEMO} - \bar{v_i}^{Klartext}}{\bar{v_i}^{Klartext}}\right)$ bzgl. Speichernutzung, Prozessorlast und Sekundärspeichernutzung (E/A)

Der Speichernutzungs-Overhead ist vernachlässigbar. Der CPU-Overhead ist wie erwartet relativ hoch, da die Verschlüsselung eine rechenintensive Aufgabe ist. Das erklärt aber den Zeitunterschied nicht, weil die absoluten Werte der CPU-Last sehr niedrig bleiben (mit Verschlüsselung ca. 0,02%). Der Ein-/Ausgabe-Overhead ist dagegen vergleichsweise deutlich höher (zwei Größenordnungen bzgl. CPU-Overhead). Ohne Verschlüsselung werden durchschnittlich ca. 400 Bytes pro Sekunde geschrieben. Mit Verschlüsselung jedoch über 23 kB/s. Das Plugin speichert also intensiv Daten auf den sekundären Speicher, was einen Flaschenhals für die Nachrichtenverarbeitung darstellt. Da der sekundäre Speicher nur beim Speichern in die Stores benutzt wird, ist der Leistungsverlust wohl durch die Wahl der Store-Implementierung zu erklären: Bei jeder Nachricht wird die Sitzung zwischen den Geräten von Alice un Bob modifiziert und da sie in der Datenbank liegt, wird beim Aktualisieren die SQLite-Datei beschrieben.

Durch die Wahl einer anderen Implementierung des lokalen Speichers könnte man die Leistung des Plugins steigern und es besser skalierbar machen. Allerdings könnte es auch reichen, wenn die Schreibfrequenz durch die Implementierung eines Datenbank-Caches verringert werden würde. Solch eine Maßnahme sollte bei der weiteren Entwicklung des Plugins berücksichtigt werden.

7 Zusammenfassung und Ausblick

In dieser Arbeit haben wir das XEP-0384 vorgestellt und mittels der Implementierung eines Plugins für *libpurple* untersucht. Daraus entstanden folgende Empfehlungen sowohl für das XEP als auch für andere XMPP-Clients und Bibliotheken, die OMEMO implementiert haben oder implementieren möchten.

- In Abschnitt 5.3.4 haben wir das Problem mit der beschränkten Eindeutigkeit der Gerät-ID vorgestellt. Eine Gerät-ID muss zwar eindeutig unter den Geräten eines bestimmten Kontaktes sein, aber nicht unter den Geräten vom sendenden *und* empfangenden Kontakt zusammen. Da es der Fall sein kann, dass Alice' Gerät die selbe ID hat wie ein Gerät von Bob, sollte das XEP vorschreiben, dass sie beim Empfang einer Nachricht *alle* Umschläge mit der gesuchten ID auf erfolgreiche Entschlüsselung überprüft, nicht nur die erste Übereinstimmung.

- Das XEP verwendet das Wort *message* in mehreren unverwandten Begriffen. Es gibt XMPP-Nachricht-Stanzas (`<message/>`), sog. Signal- und Vorschlüssel-Signal-Nachrichten ([*pre-key*] *Signal message*) und die eigentliche Textnachricht, die verschickt wird (z. B. „Hallo“). Um Missverständnisse vorzubeugen und die Spezifikation präziser zu machen, schlagen wir vor, die in OMEMO zu [*pre-key*] *Signal message* äquivalenten Konzepte umzubenennen, wobei berücksichtigt wird, dass sie in OMEMO nicht als Nachrichteninhaltbehälter sondern als Ver-/Entschlüsselungsdaten (für `<payload/>`) benutzt werden.

- Geräten wird vorgeschrieben, das Vorhandensein der eigenen ID in der Geräteliste zu prüfen, immer wenn eine PEP-Benachrichtigung von der eigenen JID ankommt. Es wird davon ausgegangen, dass das Bündel existiert. Für den Fall, bei dem ein Gerät zwar in der Geräteliste ist, aber das entsprechende Bündel im PEP-Speicher fehlt, gibt es keine Vorschriften. Dies kann z. B. passieren, wenn der PEP-Knoten des Bündels willkürlich oder versehentlich von einem zweiten Client gelöscht wird. Auch wenn dies zwar nicht die Sicherheit gefährdet, kann solch eine Situation ein Gerät unbestimmt lang unerreichbar machen. Das vorgeschriebene Verhalten sollte deshalb lauten: immer, wenn eine (neue) Version der eigenen Geräteliste über PEP eintrifft, nicht nur sie selbst zu überprüfen, sondern auch den Bündel-Knoten. Die in Bezug auf den Netzwerkverkehr günstigste Weise dies zu schaffen, ist, ein neues Bündel hochzuladen, statt es vorher abzufragen. Diesen Ansatz verfolgen wir in unserem Prototyp.

- In einem früheren Stadium des Plugins haben wir einen Umschlag für das sendende Gerät an ausgehenden Nachrichten gehängt. Da dadurch eine Sitzung mit sich selbst entsteht, bei der ständig nur die Versandkette benutzt wird und keine eingehende Nachricht jemals eintrifft, verletzt dies die *forward secrecy*,

denn die Stammkette wird nie vorangetrieben. Die OMEMO-Spezifikation sollte diese Praxis, die besonders in OpenPGP-Anwendungen verbreitet ist, explizit verbieten, und stattdessen auf andere Mechanismen für die sichere lokale Aufbewahrung von Nachrichten verweisen.

- Außerdem haben wir festgestellt, dass die Wahl eines lokalen Speichers die Geschwindigkeit der Nachrichtenverarbeitung stark beeinflussen kann und mögliche Auswege aus den dadurch verursachten Verlangsamungen vorgeschlagen.

Während der Entwicklung waren wir ständig in Kontakt sowohl mit dem Autor der OMEMO-Spezifikation als auch mit Entwicklern bestehender und neuer Implementierungen. Ihnen haben wir unsere Befunde mitgeteilt, sodass schließlich unsere XEP-Empfehlungen in den Entwurf der Version 0.2 des XEPs eingeflossen sind – die XML-Elemente für *Signal messages* heißen nun beispielsweise *envelopes*, weshalb wir das Wort „Umschlag“ in unseren Erläuterungen benutzt haben.

Mit dieser Arbeit erhoffen wir uns, zur Verbesserung der OMEMO-Spezifikation beigetragen zu haben, sodass die kommenden Versionen des XEPs in absehbarer Zeit den experimentellen Zustand verlassen und stabilere und robustere Implementierungen fördern können.

Die Zukunft von OMEMO ist noch offen. Die Verschlüsselung beliebiger Stanza-Elemente ist ein interessantes Weiterentwicklungsfeld sowie der Einsatz in Eins-zu-n-Umgebungen (z. B. Gruppenräume) und die Integration mit Medienübertragungsmechanismen (z. B. verschlüsselte Dateiübertragung). Ein weiteres Forschungsfeld ist die Untersuchung der Auswirkungen von OMEMO auf die Benutzbarkeit der Sofortnachrichtendienste und der besten Weise, auf die E2E-Verschlüsselung möglichst benutzerfreundlich den Massen vorgestellt werden kann.

In kommenden Versionen unseres Plugins werden wir einen produktiven Zustand anstreben und eventuell mit dem Entwickler eines im Laufe unserer Arbeit parallel entstandenen ähnlichen Plugins zusammenarbeiten.

8 Danksagung

Ohne die Anregung und das Interesse von Melvin Keskin, der mehrmals die Entwürfe gelesen und nützliches Feedback gegeben hat, wäre diese Arbeit nicht zustande gekommen. Vielen Dank an Andreas Straub und Daniel Gultsch, die immer offen für Vorschläge und Diskussionen über den Standard waren; an Paul Schaub, Phillip Hörist, Richard Bayerle und Eion Robb, die an der Debatte teilgenommen haben und mit denen ich Implementierungsdetails besprochen und getestet habe.

Literatur

[1] P. D'Incau. (2013, Mar.) An interview with Eugene Fooksman #erlang. [Online]. Available: https://pdincau.wordpress.com/2013/03/27/an-interview-with-eugene-fooksman-erlang/

[2] Google Inc. Open communications. [Online]. Available: https://developers.google.com/talk/open_communications

[3] H. Finney, L. Donnerhacke, J. Callas, R. L. Thayer, and D. Shaw, "OpenPGP Message Format," RFC 4880, Nov. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4880.txt

[4] T. Muldowney, *XEP-0027: Current Jabber OpenPGP Usage*, XMPP Standards Foundation, Mar. 2014, version 1.4. [Online]. Available: https://xmpp.org/extensions/xep-0027.pdf

[5] F. Schmaus, D. Schürmann, and V. Breitmoser, *XEP-0373: OpenPGP for XMPP*, XMPP Standards Foundation, Jul. 2016, version 0.1.3. [Online]. Available: https://xmpp.org/extensions/xep-0373.pdf

[6] ——, *XEP-0374: OpenPGP for XMPP Instant Messaging*, XMPP Standards Foundation, Jan. 2017, version 0.1.2. [Online]. Available: https://xmpp.org/extensions/xep-0374.pdf

[7] N. Borisov, I. Goldberg, and E. Brewer, "Off-the-record Communication, or, Why Not to Use PGP," in *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, ser. WPES '04. New York, NY, USA: ACM, 2004, pp. 77–84. [Online]. Available: https://otr.cypherpunks.ca/otr-wpes.pdf

[8] P. Saint-Andre, K. Smith, and R. Tronçon, *XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies*. O'Reilly Media, 2009.

[9] I. Paterson, P. Saint-Andre, L. Stout, and W. Tilanus, *XEP-0206: XMPP Over BOSH*, XMPP Standards Foundation, Apr. 2014, version 1.4. [Online]. Available: https://xmpp.org/extensions/xep-0206.pdf

[10] J. Hildebrand and M. Miller, *XEP-0280: Message Carbons*, XMPP Standards Foundation, Jan. 2017, version 0.11.0. [Online]. Available: https://xmpp.org/extensions/xep-0280.pdf

[11] M. Wild and K. Smith, *XEP-0313: Message Archive Management*, XMPP Standards Foundation, Mar. 2016, version 0.5.1. [Online]. Available: https://xmpp.org/extensions/xep-0313.pdf

[12] P. Saint-Andre, *Extensible Messaging and Presence Protocol (XMPP): Core*, RFC 6120, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc6120.txt

[13] ——, *XEP-0001: XMPP Extension Protocols*, XMPP Standards Foundation, Mar. 2010, version 1.20. [Online]. Available: https://xmpp.org/extensions/xep-0001.pdf

[14] P. Saint-Andre and K. Smith, *XEP-0163: Personal Eventing Protocol*, XMPP Standards Foundation, Jul. 2010, version 0.1. [Online]. Available: https://xmpp.org/extensions/xep-0163.html

[15] A. Straub, *XEP-0384: OMEMO Encryption*, XMPP Standards Foundation, Feb. 2017, version 0.2. [Online]. Available: https://xmpp.org/extensions/xep-0384.pdf

[16] S. Turner, A. Langley, and M. Hamburg, "Elliptic curves for security," RFC 7748, Jan. 2016. [Online]. Available: https://rfc-editor.org/rfc/rfc7748.txt

[17] D. A. McGrew and J. Viega, "The galois/counter mode of operation (GCM)," 2004. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf

[18] ——, "The security and performance of the galois/counter mode (GCM) of operation," in *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings*, 2004, pp. 343–355. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30556-9_27

[19] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976.

[20] M. Marlinspike, *The X3DH Key Agreement Protocol*, Nov. 2016, revision 1. [Online]. Available: https://whispersystems.org/docs/specifications/x3dh/x3dh.pdf

[21] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography*, ser. Lecture Notes in Computer Science, vol. 3958. Springer, 2006, pp. 207–228. [Online]. Available: https://cr.yp.to/ecdh/curve25519-20060209.pdf

[22] M. Marlinspike, *The Double Ratchet Algorithm*, Nov. 2016, revision 1. [Online]. Available: https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf

[23] "What is libpurple?" Mar. 2015. [Online]. Available: https://developer.pidgin.im/wiki/WhatIsLibpurple

[24] S. Verschoor, *OMEMO: Cryptographic Analysis Report*, Amsterdam, Jun. 2016, version 1.0. [Online]. Available: https://conversations.im/omemo/audit.pdf

[25] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz, "How secure is TextSecure?" in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, 2016, pp. 457–472. [Online]. Available: https://eprint.iacr.org/2014/904.pdf

[26] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the Signal Messaging Protocol," *IACR Cryptology ePrint Archive*, vol. 2016, p. 1013, 2016. [Online]. Available: https://eprint.iacr.org/2016/1013.pdf

[27] D. H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Feb. 1997. [Online]. Available: https://rfc-editor.org/rfc/rfc2104.txt

[28] G. Allen and M. Owens, *The Definitive Guide to SQLite*, 2nd ed. Berkely, CA, USA: Apress, 2010.

[29] M. Wild, *XEP-0334: Message Processing Hints*, XMPP Standards Foundation, Sep. 2015, version 0.2. [Online]. Available: https://xmpp.org/extensions/xep-0334.pdf

[30] E. G. Peyrot, *XEP-0380: Explicit Message Encryption*, XMPP Standards Foundation, Oct. 2016, version 0.1. [Online]. Available: https://xmpp.org/extensions/xep-0380.pdf

[31] S. E. Ahmed and R. J. McIntosh, "An asymptotic approximation for the birthday problem," *Crux Mathematicorum*, vol. 26, no. 3, pp. 151–155, Apr. 2000. [Online]. Available: https://cms.math.ca/crux/v26/n3/page151-155.pdf