1984not Security Whitepaper

Max Bruckner, Dipl.-Ing. Bernd Herzmann

17.03.2019

Inhaltsverzeichnis

1 Einführung

Für 1984not ist Ihre Privatsphäre erste Priorität. Dieses Dokument verfolgt den Zweck, die dafür ergriffenen technischen Maßnahmen, insbesondere die Ende-Zu-Ende-Verschlüsselung in der Übersicht und im Detail zu erklären.

Herfür sollen zuerst einige Grundlegende kryptografische Konzepte Eingeführt werden und anschliessend die Verwendung derer in 1984not's Kurznachrichtendienst.

1.1 Grundlegende Konzepte

Dieser Abschnitt soll Ihnen einen vereinfachten Überblick über grundlegende hauptsächlich kryptografische Konzepte verschaffen um den Rest dieses Whitepapers besser nachvollziehne zu können.

1.1.1 Sicherheit, Komplexität und Offenheit

Ein Sicherheitskonzept oder eine Verschlüsselungsmethode kann immer Fehler enthalten, die bis dato niemand entdeckt hat oder (im schlimmsten Fall) bereits entdeckt wurden und im Geheimen (z.B. von Geheimdiensten) bereits ausgenutzt werden. Deshalb ist es wichtig, dass möglichst viele Personen dieses auf Fehler untersuchen und etwaige fehler melden, sodass sie behoben werden können. Eine essentielle Vorraussetzung dafür ist, dass ein System nicht unnötig komplex ist. Dass also die kleinstmögliche Komplexität gewählt wird, um ein gegebenes Problem zu lösen, weil es dadurch schwieriger wird, dass Probleme übersehen werden.

Solch ein Sicherheitskonzept oder eine Verschlüsselungsmethode, so sicher sie auch sein mag, kann weiterhin nur dann die theoretische Sicherheit garantieren wenn das Konzept fehlerfrei in die Praxis umgesetzt wird, was wiederum ein Argument dafür ist, die Komplexität so gering wie möglich zu halten.

Um aber überhaupt erst zu ermöglichen, dass Fehler gefunden werden, muss sichergestellt werden, dass das Design des Systems offen dokumentiert ist, sodass andere Leute dieses Analysieren und auf Fehler hinweisen können. Diese Aufgabe wird zum Teil von diesem Whitepaper übernommen.

1.1.2 Nachrichtenübertragung über das Internet

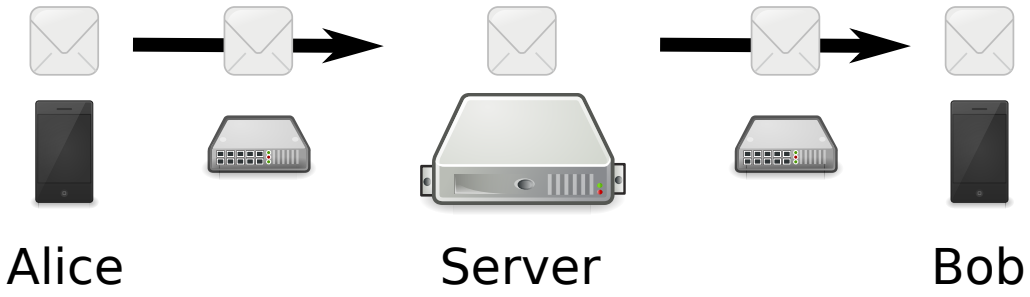

Abbildung 1: Server-Modell (CC-BY-SA)

Werden Nachrichten über das Internet übertragen, so erfolgt dies immer über eine oder mehrere Zwischenstationen. Prinzipbedingt können diese Zwischenstationen die vollständigen Daten der übertragenen Nachricht mitlesen. An dieser Stelle setzt Verschlüsselung ein.

Hierbei gilt es zwischen Zwischenstationen zum Transport (z.B. Router, bei Internet Service Provider) und dem Server des Diensteanbieters zu unterscheiden.

1.1.3 Transportverschlüsselung

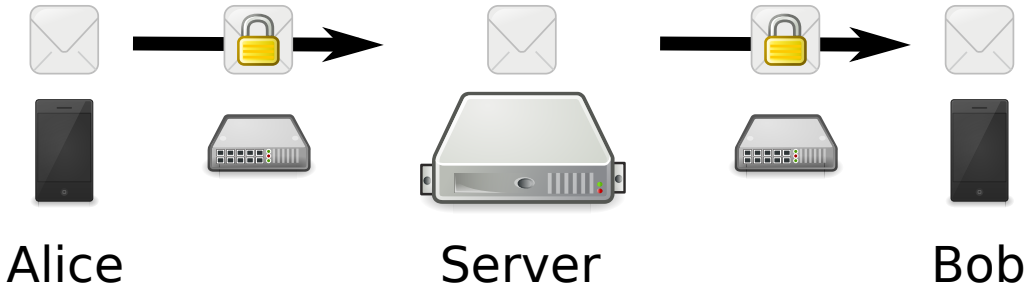

Abbildung 2: Transportverschlüsselung (CC-BY-SA)

Hier verschlüsselt Alice ihre Nachricht und sendet sie an den Server. Dort wird die Nachricht entschlüsselt, und nach erneuter Verschlüsselung an Bob verschickt, der sie wiederum entschlüsselt um sie zu lesen.

Auf dem Transportweg von Alice zum Server und vom Server zu Alice können die Zwischenstationen die Nachricht nicht lesen, daher Transportverschlüsselung.

Der Server hat volle Einsicht in die Nachricht und kann sie beliebig manipulieren.

1.1.4 Ende-Zu-Ende-Verschlüsselung

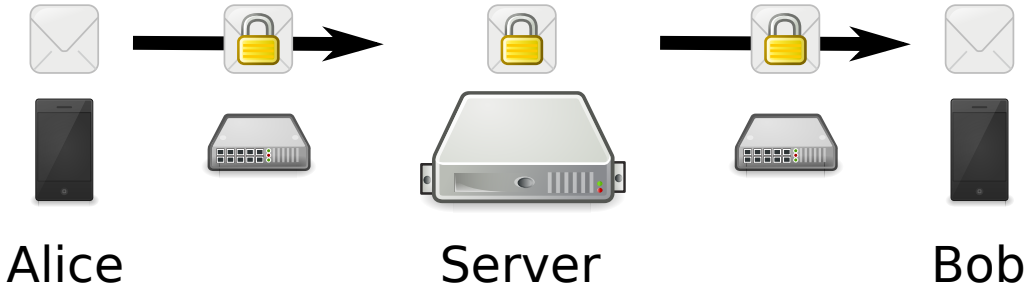

Abbildung 3: Ende-Zu-Ende Verschlüsselung (CC-BY-SA)

Hier verschlüsselt Alice ihre Nachricht, sendet sie an den Server, wo sie, ohne die Verschlüsselung zu entfernen, direkt an Bob weitergeleitet wird. Bob entschlüsselt sie wiederum, um sie zu lesen.

Dies bedeutet, dass Alice und Bob die einzigen Personen sind, die den Inhalt der Nachricht sehen können.

In der Praxis wird dies oft zusätzlich mit Transportverschlüsselung kombiniert, so auch bei 1984not.

1.1.5 Verschlüsselung und Authentifizierung

Verschlüsselung ist das Konzept eine Nachricht (Klartext) mithilfe eines Schlüssels so in einen Ciphertext zu transformieren (verschlüsseln), dass der Klartext nur mit einem passenden Schlüssel wieder zurück in Klartext transformiert (entschlüsselt) werden kann.

Verschlüsselung alleine ist aber nicht ausreichend für eine sichere Kommunikation. Der Inhalt einer Nachricht soll nicht nur geheim gehalten werden, sondern es soll auch sichergestellt werden, dass eine Nachricht tatsächlich von dem erwünschten Gesprächspartner stammt.

An dieser Stelle ein Beispiel: Viele Verschlüsselungsverfahren haben die Eigenschaft, dass eine Erhöhung des Wertes eines Zeichens im Ciphertext 1 zu 1 einer Erhöhung des Wertes im Klartext

entspricht. Mit anderen Worten: Es ist Möglich gezielt den Klartext zu manipulieren, ohne ihn zu kennen. Mal angenommen Alice gibt ihrem Banker Bob den Auftrag, Charlie 1000€ zu schicken. Die Nachricht hat ein Standardisiertes Format, sodass Charlie genau weiß an welcher Position in der Nachricht sich der Geldbetrag befindet. Charlie kann nun einfach an dieser Position den Ciphertext um 1 erhöhen und plötzlich steht in dem entschlüsselten Klartext 2000€ statt 1000€.

Nachrichten-Authentifizierung löst dieses Problem, indem die Kenntnis eines Schlüssels benötigt wird um den Inhalt einer Nachricht abzusegnen. Der Empfänger kann ebenfalls mit einem Schlüssel prüfen, ob die Empfangene Nachricht korrekt abgesegnet wurde.

In der Praxis wird hierfür zumeist entweder eine Signatur oder ein Message Authentication Code (MAC) verwendet, siehe unten.

1.1.6 Hash-Funktionen

Bei einer kryptografischen Hash-Funktion handelt es sich um eine Funktion, die aus einer Eingabe beliebiger Länge (in der Form von Bytes) auf eine Ausgabe fester Länge abbildet. Diese Ausgabe bezeichnet man als den *Hash* der Ausgabe.

Beispiel: H("`Hi Alice`") → `ee9c2da9ff975212f4a4ff5cf8bfb95e25e77cf3f3bf4c715c0ea06299b7060a`

Sie haben außerdem unter anderem folgende Eigenschaften:

Unumkehrbarkeit Es ist nicht möglich von dem Hash auf die Eingabe zurückzuschließen, es sei denn man probiert so lange Eingaben durch, bis man die passende gefunden hat.

Kollisionsresistenz Es ist so unwahrscheinlich zwei unterschiedliche Eingaben zu finden, die den gleichen Hash haben, dass man es in der Praxis als unmöglich annehmen kann.

Unkorreliertheit Die Beziehung in der zwei Eingaben stehen darf aus deren Hashes nicht hervorgehen.

1.1.7 Symmetrische Verschlüsselung

Bei symmetrischer Verschlüsselung wird für Ver- und Entschlüsselung der gleiche Schlüssel verwendet. Das bedeutet dass sowohl Sender als auch Empfänger denselben Schlüssel benötigen, um verschlüsselt Nachrichten auszutauschen. Eine Herausforderung ist hierbei der Schlüsselaustausch, also wie der Schlüssel von Alice zu Bob übertragen werden kann, ohne dass er dabei abgehört werden kann. Dieses Problem wird durch Asymmetrische Verschlüsselung gelöst.

1.1.8 Message Authentication Codes (MAC)

Message Authentication Codes sind eine Art der Nachrichten-Authentifizierung mit symmetrischem Schlüssel. Hierfür kommt in der Regel eine Hash-Funktion mit einem Schlüssel zum Einsatz. Ein MAC wird mithilfe der Nachricht (in der Regel als Ciphertext) und dem Schlüssel erzeugt. Anschließend werden MAC und Nachricht gemeinsam übertragen.

Der Empfänger nutzt seinen Schlüssel um Ebenfalls den MAC zu berechnen. Ist dieser identisch mit dem mitgelieferten MAC, so ist die Nachricht gültig. Würde ein Angreifer die Nachricht verändern, so würde sich beim Empfänger der MAC ändern. Der Angreifer müsste also auch den MAC auf die neue Nachricht anpassen. Dies ist jedoch nicht möglich, da hierfür der Schlüssel benötigt wird.

Hier sei auch nochmal darauf hingewiesen, dass sowohl Empfänger als auch Absender der Nachricht in der Lage sind, einen gültigen MAC zu erzeugen. Es lässt sich also nicht nachweisen, wer die Nachricht authentifiziert hat. Dadurch wird Abstreitbarkeit ermöglicht.

1.1.9 Asymmetrische Verschlüsselung

Bei asymmetrischer Verschlüsselung haben Alice und Bob jeweils ein Paar von Schlüsseln: Ein öffentlicher und ein privater Schlüssel. Eine Nachricht wird mit dem öffentlichen Schlüssel eines Paares verschlüsselt, kann aber nur mit dem privaten Schlüssel wieder entschlüsselt. Wegen dieser assymmetrie (Die Nachricht wird mit einem anderen Schlüssel entschlüsselt, als sie verschlüsselt wurde) redet man von asymmetrischer Verschlüsselung.

Möchten Alice und Bob Nachrichten miteinander austauschen, so tauschen sie erst unverschlüsselt ihre öffentlichen Schlüssel aus. Anschließend kann Alice Nachrichten mit Bobs öffentlichem Schlüssel verschlüsseln, der sie dann mit seinem privaten Schlüssel entschlüsseln kann und umgekehrt.

Dies vereinfacht den Schlüsselaustausch immens, da die öffentlichen Schlüssel im Gegensatz zu einem symmetrischen Schlüssel nicht vor neugierigen Blicken geschützt werden müssen.

In der Praxis wird assymetrische Verschlüsselung aus diversen Gründen jedoch nicht für die Verschlüsselung der eigentlichen Nachrichten eingesetzt, sondern lediglich um einen geheimen symmetrischen Schlüssel zu übertragen, mit dem die eigentlichen Nachrichten verschlüsselt werden.

1.1.10 Man-In-The-Middle-Angriff

Eine Verbindung zwischen Alice und Bob über das Internet geschieht niemals direkt, sondern immer über Zwischenstationen wie z.B. WLAN-Router, Internet Provider etc.. Das bedeutet wiederum, dass man darauf angewiesen ist, dass jede dieser Zwischenstationen sich korrekt verhält.

Wird eine der Zwischenstationen von einem Angreifer (dem Mann in der Mitte) kontrolliert, so ermöglicht dies eine Reihe von Angriffen. Unter anderem:

- Der Ciphertext übertragener Nachrichten könnte zur späteren Analyse abgespeichert werden.
- Nachrichten könnten blockiert werden, sodass sie auf der Gegenseite nicht ankommen.
- Gespeicherte Nachrichten könnten mehrfach gesendet werden (replay). Hier müssen Vorkehrungen getroffen werden, dass solche doppelt gesendeten Nachrichten nicht als gültig anerkannt werden. Es wäre beispielsweise wohl kaum erwünscht, wenn ein Angreifer eine Überweisung mehrfach ausführen könnte, indem er die Nachricht mehrfach schickt. 1984not beugt dem im Protokoll vor.
- Nachrichten könnten aktiv manipuliert werden. Authentifizierung schützt gegen dieses Szenario.
- Beim Austausch von öffentlichen Schlüsseln kann der Angreifer den Schlüssel durch seinen eigenen austauschen und somit später alle gesendeten Nachrichten problemlos entschlüsseln. Um dies zu verhindern sollten öffentliche Schlüssel manuell verifiziert werden, z.B. per QR-Code.

1.1.11 Signaturen

Signaturen sind eine Art der Nachrichten-Authentifizierung mit asymmetrischem Schlüssel. Ähnlich wie bei assymetrischer Verschlüsselung existieren immer Paare aus einem öffentlichen und einem privaten Schlüssel. Eine Signatur einer Nachricht wird mithilfe des privaten Schlüssels und der Nachricht berechnet und ähnlich einem MAC mit der Nachricht übertragen. Der Empfänger prüft die Gültigkeit einer Signatur mithilfe des öffentlichen Schlüssels. Würde ein Angreifer die Nachricht verändern, so bräuchte er den privaten Schlüssel um auch die Signatur anzupassen.

Was Signaturen im Gegensatz zu MACs nicht bieten ist Abstreitbarkeit. Eine Signatur verhält sich wie eine Unterschrift. Signiert man eine Nachricht, so ist dies ein Beweis, dass der Autor der Nachricht im Besitz des privaten Schlüssel ist. Wie bei einem unterzeichneten Vertrag, kann einem nachgewiesen werden, dass man die Nachricht signiert hat (sofern der Schlüssel nicht abhanden gekommen ist).

1.1.12 Diffie-Hellman-Schlüsselaustausch

Der Diffie-Hellman-Schlüsselaustausch ist eine Methode um mit zwei assymetrischen Schlüsselpaaren einen gemeinsamen symmetrischen Schlüssel herleiten zu können. Dies ist hilfreich damit zwei Gesprächspartner, die bereits ihre öffentlichen Schlüssel ausgetauscht haben, sich symmetrisch verschlüsselte Nachrichten schicken können.

Der Schlüsselaustausch geschieht mit einer sogenannten Diffie-Hellman-Funktion. Mal angenommen die Gesprächspartner seien Alice und Bob mit jeweils eigenem Schlüsselpaar, so berechnet die Diffie-Hellman-Funktion den gleichen geteilten Schlüssel (*shared key*) wenn man ihr Alice' privaten und Bobs öffentlichen Schlüssel, oder aber wenn man ihr Bobs privaten und Alice' öffentlich Schlüssel übergibt:

$$\texttt{shared\_key} = \texttt{DH}\left(\texttt{alice\_private}, \texttt{bob\_public}\right) = \texttt{DH}\left(\texttt{bob\_private}, \texttt{alice\_public}\right)$$

Den geteilten Schlüssel erhalten Alice und Bob also jeweils, indem sie die Diffie-Hellman Funktion auf ihren eigenen privaten Schlüssel und den öffentlichen Schlüssel des Gesprächspartners anwenden.

1.1.13 Forward Secrecy

Forward-Secrecy ist eine Eigenschaft eines Kryptosystemes für den Fall, dass der Ciphertext von übertragenen Nachrichten mitgeschnitten und aufbewahrt wurden ("Voratsdatenspeicherung"[1]).

Hat ein Kryptosystem **keine** Forward-Secrecy, so kann ein Angreifer selbst Jahre nach dem Mitschneiden des Ciphertextes einer Nachricht noch den Schlüssel von einem der Gesprächspartner entwenden und damit alle mitgeschnittenen Nachrichten nachträglich entschlüsseln.

Mit Forward-Secrecy werden alle Nachrichten nur mit temporären Schlüsseln verschlüsselt, die nach erfolgreicher Benutzung wieder gelöscht werden. Dies macht ein nachträgliches entschlüsseln von Mitschnitten unmöglich.

1.1.14 Key Derivation

Key Derivation ist das Ableiten von mehreren Unterschlüsseln (*subkeys*) aus einem Hauptschlüssel (*main key*).

Mithilfe eines Hauptschlüssels und einer Unterschlüsselnummer (*subkey counter*) erzeugt eine Key Derivation Function (KDF) einen Unterschlüssel:

$$\texttt{subkey} = \texttt{KDF}\left(\texttt{main\_key}, \texttt{subkey\_counter}\right)$$

Dieser Unterschlüssel hat die folgenden Eigenschaften hat:

Unumkehrbarkeit Es ist nicht möglich von einem Unterschlüssel auf den Hauptschlüssel zurückzuschließen, es sei denn man probiert so lange Hauptschlüssel durch, bis man den passenden gefunden hat.

Unkorreliertheit Man sieht den verschiedenen Unterschlüsseln nicht an, dass sie aus dem gleichen Hauptschlüssel hervorgegangen sind.

Diese Eigenschaften gleichen der einer Hash-Funktion, dementsprechend könnte man beispielsweise eine KDF mithilfe einer Hash-Funktion implementieren:

$$\texttt{KDF}\left(\texttt{main\_key}, \texttt{subkey\_counter}\right) = \texttt{H}\left(\texttt{main\_key}||\texttt{subkey\_counter}\right)$$

[1] Nicht zu verwechseln mit der Vorratsdatenspeicherung der deutschen Bundesregierung

1.1.15 Abstreitbarkeit (Plausible Deniability)

Abstreitbarkeit beschäftigt sich mit dem Thema, ob die verwendete Kryptographie Beweise hinterlässt, die theoretisch z.B. vor Gericht gegen einen verwendet werden könnten. Dieses Problem entsteht aus der Notwendigkeit, Nachrichten zu authentifizieren.

Verschickt man Nachrichten mit einer Signatur, so lässt sich in der Regel nachweisen, dass man diese Nachricht verfasst hat, es lässt sich nicht abstreiten. Bei MACs hingegen geht nicht hervor, ob der MAC vom Absender oder vom Empfänger erstellt wurde. Es steht Aussage gegen Aussage, da der Empfänger die Nachricht hätte fälschen können.

Wichtig: Selbst wenn Abstreitbarkeit gegeben ist, so ist das natürlich kein Garant, dass einen Nachrichten vor Gericht nicht trotzdem belasten. So werden beispielsweise regelmäßig Verbrecher nur auf Basis von unverschlüsselten und unauthentifizierten Email-Mitschnitten verurteilt.

2 Eingesetzte Kryptographie in 1984not

2.1 Nachrichten-Verschlüsselung

Um Chat-Nachrichten zu Verschlüsseln setzt 1984not auf das sogenannte Double-Ratchet[2] von OpenWhisper Systems, das sich mittlerweile als Quasi-Standard unter Krypto-Messengern etabliert hat.

Eine vollständige Beschreibung (englisch) findet sich in der Spezifikation[2].

Das Double-Ratchet bietet nicht nur Forward Secrecy und Abstreitbarkeit, sondern hat auch eine selbstheilende Eigenschaft, sodass in einigen Fällen von kompromittierten Schlüsseln die Verschlüsselung vollständig wiederhergestellt werden kann.

1984not verwendet das Double-Ratchet in der Variante mit Header-Verschlüsselung.

2.1.1 Überblick über das Double-Ratchet

Im Folgenden soll ein kurzer Überblick über das Double-Ratchet gegeben werden. Weitere Informationen entnehmen Sie der Spezifikation[2]. Hierfür werden die zwei Hauptkkonzepte des Ratchet erläutert: Symmetrisches-Ratchet und Diffie-Hellman-Ratchet.

Ein Ratchet ist hier eine Abfolge von immer wiederkehrenden Schritten, die jeweils neue Schlüssel erzeugen.

Symmetrisches Ratchet

Zuallererst eine Liste von Schlüsseln, die im symmetrischen Ratchet eingesetzt werden:

Root-Key Ein geteilter “Wurzel”-Schlüssel den beide Gesprächspartner mithilfe des Diffie-Hellman-Ratchets erhalten und immer am Beginn eines symmetrischen Ratchets steht.

Message-Key Ein Nachrichten-Schlüssel der zum Ver-/Entschlüsseln von genau einer Nachricht verwendet wird.

Chain-Key Ein “Ketten”-Schlüssel aus welchem im Verlauf des symmetrischen Ratchets weitere Chain-Keys und Message-Keys abgeleitet werden.

Das symmetrische Ratchet stellt sicher, dass jede Nachricht mit einem eigenen symmetrischen Schlüssel verschlüsselt wird, dem Message-Key. Das symmetrische Ratchet startet mit einem Root-Key, aus dem mit einer Key Derivation Function (KDF) ein Chain-Key abgeleitet wird. Dieser Root-Key ist bei beiden Gesprächspartnern gleich, sodass das Ratchet sowohl für Alice als auch für Bob identisch abläuft.

Jeder Ratchet-Schritt des symmetrischen Ratchets sieht nun folgendermassen aus:

1. Ein Message-Key wird aus dem aktuellen Chain-Key abgeleitet. Dieser Message Key wird verwendet um eine Nachricht zu Verschlüsseln (beim Senden) oder zu entschlüsseln (beim Empfangen). Nach dem Verschlüsseln/Entschlüsseln wird der Message-Key gelöscht.

2. Der nächste Chain-Key wird aus dem aktuellen Chain-Key abgeleitet und anschliessend der aktuelle Chain-Key gelöscht.

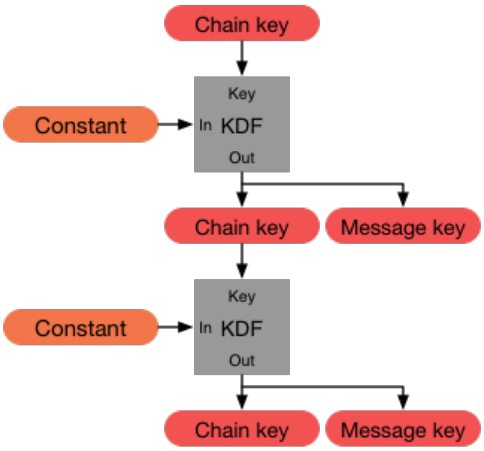

Abbildung 4: Symmetrisches Ratchet (Public Domain)

Solch ein Ratchet-Schritt wird immer durchlaufen wenn eine Nachricht gesendet oder empfangen wird. Für jedes Gespräch existieren pro Gesprächspartner zu jedem Zeitpunkt jeweils zwei symmetrische Ratchets. Eines zum Empfangen und eines zum Senden von Nachrichten. Message-Keys können auch abgespeichert werden, wenn die zugehörige Nachricht noch nicht angekommen ist. Das ist nützlich wenn Nachrichten nicht in der richtigen Reihenfolge ankommen.

Dieses Symmetrische-Ratchet bietet Forward Secrecy, weil die Message Keys und alten Chain Keys nach ihrer Verwendung unwiderbringlich gelöscht werden, weshalb sie nicht nachträglich kompromittiert werden können. Würde jedoch einer der Chain-Keys kompromittiert, so sind damit auch alle zukünftigen Message und Chain Keys dieses Ratchets kompromittiert. Um die Auswirkungen dieses Szenarios zu verhindern wird ein symmetrisches Ratchet immer nur so lange benutzt, bis der andere (empfangende) Gesprächspartner mit einer Nachricht antwortet, sich also die "Richtung" des Gespräches ändert. Findet solch ein Richtungswechsel statt, so wird mithilfe des Diffie-Hellman-Ratchets ein neues symmetrisches Ratchet begonnen.

Diffie-Hellman-Ratchet

Hier ein Überblick der im Diffie-Hellman-Ratchet verwendeten Schlüssel:

Sende-Schlüsselpaar Ein temporäres asymmetrisches Schlüsselpaar aus dem ein neues symmetrisches Ratchet zum Senden von Nachrichten hergeleitet wird. Dieses wird vom sendenden Gesprächsteilnehmer am Anfang des Diffie-Hellman-Ratchet-Schrittes erzeugt.

Root-Key Jeder neue Diffie-Hellman-Ratchet-Schritt wird mit der Ableitung eines neuen Root-Keys eingeleitet, der bei beiden Gesprächspartnern identisch ist und auf dem das symmetrische Ratchet aufbaut.

Öffentlicher Empfangs-Schlüssel Der öffentliche Teil des Sende-Schlüsselpaars des anderen Gesprächspartners. Diesen Schlüssel erhält der Empfänger aus den empfangenen Nachrichten.

Wie soeben erläutert findet der nächste Schritt des Diffie-Hellman-Ratchet immer dann statt, wenn sich die Richtung des Gespräches ändert, also Alice auf Bob's Nachrichten antwortet oder umgekehrt. Dies ist vergleichbar mit einem Ball, den sich Alice und Bob zuwerfen. Wer gerade den Ball in der Hand hält generiert bei der ersten nachfolgend gesendeten Nachricht ein neues Sende-Schlüsselpaar, startet ein neues symmetrisches Ratchet und wirft den Ball dem anderen zu, mitsamt dem öffentlichen Sende-Schlüssel.

Hier die Teilschritte:

- **Sender**
  1. Der Sender generiert ein neues Sende-Schlüsselpaar.
  2. Aus dem Sende-Schlüsselpaar, dem öffentlichen Empfangs-Schlüssel und dem aktuellen Root-Key wird ein neuer Root-Key abgeleitet.
  3. Mithilfe des neuen Root-Key wird ein neues symmetrisches Ratchet gestartet und damit Nachrichten versendet.
- **Empfänger**
  1. Der Empfänger empfängt eine Nachricht und erhält dadurch den Öffentlichen Empfangs-Schlüssel.
  2. Aus dem alten Sende-Schlüsselpaar, dem öffentlichen Empfangs-Schlüssel und dem aktuellen Root-Key wird ein neuer Root-Key abgeleitet.
  3. Mithilfe des neuen Root-Key wir ein neues symmetrisches Ratchet gestartet und damit die Nachricht entschlüsselt.

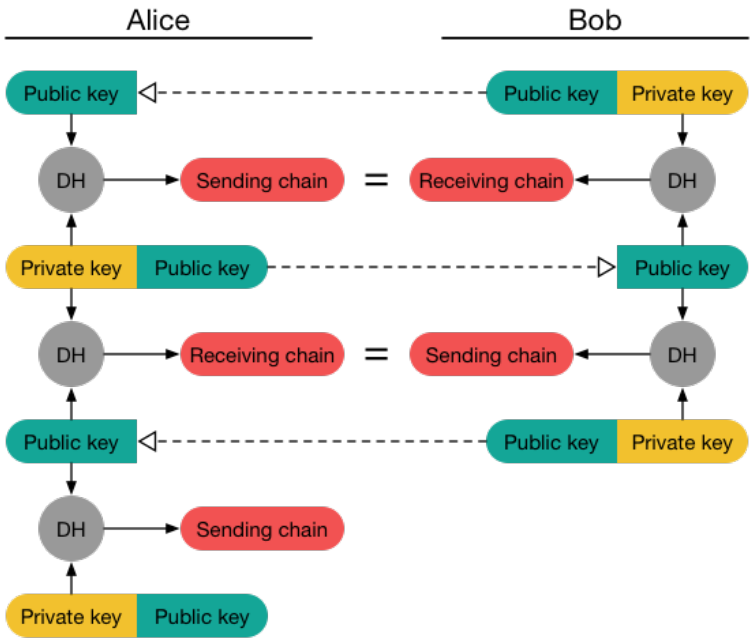

Abbildung 5: Diffie-Hellman-Ratchet (Public Domain)

2.1.2 Implementierung der Diffie-Hellman-Funktion

Die in der Implementierung des Double-Ratchet verwendete Diffie-Hellman-Funktion für Curve25519-Schlüssel wurde mithilfe von `libsodiums` Funktionen `crypto_scalarmult` und `crypto_generichash` implementiert. Hierbei handelt es sich um X25519 (ECDH über Curve25519) und die Blake2b Hash-Funktion.

Die Implementierung bestimmt mit `sodium_compare`, welcher öffentliche Schlüssel ein größere Little-Endian Zahl ist.

Gegeben seien ein asymmetrisches Schlüsselpaar (our) und ein öffentlicher Schlüssel (their), hiermit ist die Diffie-Hellman-Funktion implementiert als:

$$\mathtt{DH}\left(\mathtt{our_s}, \mathtt{their_p}\right) = \begin{cases} \mathtt{H}(\mathtt{X25519}(\mathtt{our_s}, \mathtt{their_p}) || \mathtt{our_p} || \mathtt{their_p}) & \text{if } \mathtt{our_p} > \mathtt{their_p} \\ \mathtt{H}(\mathtt{X25519}(\mathtt{our_s}, \mathtt{their_p}) || \mathtt{their_p} || \mathtt{our_p}) & \text{if } \mathtt{our_p} < \mathtt{their_p} \end{cases}$$

Hierbei kennzeichnet s einen privaten (*secret*) Schlüssel und p einen öffentlichen (*public*).

2.1.3 Ableitung des Master-Keys

Um einen Root-Key abzuleiten wird ein vorheriger Root-Key benötigt, das ist aber offensichtlich nicht möglich um den ersten Root-Key abzuleiten. Hierfür wird zuerst bestimmt, welcher der beiden Gesprächspartner die Rolle von Alice und welcher die Rolle von Bob übernimmt. Dies geschieht mithilfe eines vergleiches der öffentlichen Identitäts-Schlüssel mit `libsodiums` `sodium_compare`. Der Gesprächspartner mit in Little-Endian größerem öffentlichem Schlüssel sei Alice, die andere Person sei Bob.

Nun wird mithilfe des sogenannten Triple-Diffie-Hellman ein Hauptschlüssel (Master Key) abgeleitet, der als Schlüssel für die Key-Derivation der ersten Root-, Chain- und Header-Keys verwendet wird. Für das Triple-Diffie-Hellman werden die Identitäts-Schlüsselpaare von Alice und Bob sowie jeweils ein Kurzzeit-Schlüsselpaar (ephemeral Keypair) verwendet. Mit diesen vier Schlüsselpaaren wird in verschiedenen Kombinationen ein Diffie-Hellman berechnet und diese aneinandergereiht in eine Hash-Funktion gespeist, um den Master Key zu erhalten.

Auf Alice' Seite:

$$\mathtt{DH_1} = \mathtt{DH}\left(\mathtt{alice\_private\_identity}, \mathtt{bob\_public\_ephemeral}\right)$$

$$\mathtt{DH_2} = \mathtt{DH}\left(\mathtt{alice\_private\_ephemeral}, \mathtt{bob\_public\_identity}\right)$$

$$\mathtt{DH_3} = \mathtt{DH}\left(\mathtt{alice\_private\_ephemeral}, \mathtt{bob\_public\_ephemeral}\right)$$

$$\mathtt{master\_key} = \mathtt{H}\left(\mathtt{DH_1} || \mathtt{DH_2} || \mathtt{DH_3}\right)$$

Auf Bobs Seite:

$$\mathtt{DH_1} = \mathtt{DH}\left(\mathtt{bob\_private\_ephemeral}, \mathtt{alice\_public\_identity}\right)$$

$$\mathtt{DH_2} = \mathtt{DH}\left(\mathtt{bob\_private\_identity}, \mathtt{alice\_public\_ephemeral}\right)$$

$$\mathtt{DH_3} = \mathtt{DH}\left(\mathtt{bob\_private\_ephemeral}, \mathtt{alice\_public\_ephemeral}\right)$$

$$\mathtt{master\_key} = \mathtt{H}\left(\mathtt{DH_1} || \mathtt{DH_2} || \mathtt{DH_3}\right)$$

Die Hash-Funktion ist hierbei Blake2b implementiert durch `libsodiums` `crypto_generichash`.

Die Identitäts-Schlüsselpaare werden bei der ersten Nutzung der 1984not-App erzeugt und über 1984not's Identitäts-Server ausgetauscht. Das Kurzzeit-Schlüsselpaar des Empfängers der ersten Nachricht wird vom Sender aus einer Liste von vorgenerierten Prekeys ausgewählt, siehe unten. Das Kurzzeit-Schlüsselpaar des Senders wird vor dem Senden der ersten Nachricht generiert.

2.1.4 Format einer Nachricht

Um Nachrichten zu enkodieren wird das Format Google Protocoll Buffers[8] verwendet. Das Nachrichten-Format ist in einer Menschenlesbaren Definition definiert und wird von der Bibliothek `protobuf-c` in Binärdaten zur Übertragung umgewandelt.

Das Paket einer Nachricht hat einen hierarchischen Aufbau. An oberster Stelle befinden sich der Paketheader, der verschlüsselte Double-Ratchet-Header und die verschlüsselte Nachricht.

Die verschlüsselte Nachricht wird mit dem aktuellen Message-Key und der Message-Nonce, die im Paket-Header enthalten ist, verschlüsselt. Der verschlüsselte Double-Ratchet-Header wird mit der Header-Nonce, die ebenfalls im Paket-Header enthalten ist, verschlüsselt. Hierfür kommt jeweils `libsodiums` Funktion `crypto_secretbox_easy` zum Einsatz. Implementiert ist diese Funktion mit der XSalsa20 Stromverschlüsselung für die Verschlüsselung und Poly1305 als MAC.

Die Nonces werden mithilfe von `libsodiums` Funktion `randombytes_buf` aus Zufallszahlen generiert.

Listing 1: Paket

```
syntax = "proto2";
import "packet_header.proto";

message Packet {
        required PacketHeader packet_header = 1;
        optional bytes encrypted_axolotl_header = 2;
        optional bytes encrypted_message = 3;
}
```

Der Paketheader enthält hierbei Informationen über die Protokoll-Version, den Typ der Nachricht und die kryptografischen Nonces für die Verschlüsselung des Headers und der Nachricht. Im Falle einer Prekey-Nachricht enthält der Paketheader zusätzlich den öffentliche Identitäts-Schlüssel des Absenders, den öffentlichen Kurzzeitschlüssel des Absenders und den öffentlichen Prekey des Empfängers, der für diese Konversation ausgewählt werden soll.

Listing 2: Paket-Header

```
syntax = "proto2";
message PacketHeader {
        required uint32 current_protocol_version = 1;
        required uint32 highest_supported_protocol_version = 2;
        enum PacketType {
                PREKEY_MESSAGE = 0;
                NORMAL_MESSAGE = 1;
        }
        optional PacketType packet_type = 3 [default = NORMAL_MESSAGE];
        optional bytes header_nonce = 4;
        optional bytes message_nonce = 5;
        optional bytes public_identity_key = 16; //only prekey messages
        optional bytes public_ephemeral_key = 17; //only prekey messages
        optional bytes public_prekey = 18; //only prekey messages
}
```

Der Double-Ratchet-Header enthält den aktuellen Kurzzeit-Schlüssel des Absenders, die Nummer der aktuellen Nachricht und die Nummer der letzten Nachrichten im vorherigen symmetrischen Ratchet.

Listing 3: Double-Ratchet-Header

```
syntax = "proto2";
message Header {
        optional bytes public_ephemeral_key = 1;
```

```
4           //fixed32 in order to not leak data from the length
5           optional fixed32 message_number = 2;
6           optional fixed32 previous_message_number = 3;
7   }
```

2.1.5 Verstecken der Länge von Nachrichten

Um den Inhalt von verschlüsselten Nachrichten zu schützen reicht es oft nicht aus, den Inhalt zu verschlüsseln, sondern es muss auch die Länge der Nachricht versteckt werden. Warum das so ist, soll hier anhand von einem Beispiel erklärt werden:

Mal angenommen Alice schickt eine Nachricht an Bob und die Nachricht ist zwei Zeichen lang. Und angenommen ein Angreifer, Eve, weiß nur die Länge der Nachricht, hat aber sonst keinerlei Informationen. Theoretisch gäbe es nun $256^2 = 65536$ Möglichkeiten, was die beiden Zeichen der Nachricht sein könnten. Aber da Alice wahrscheinlich nur Buchstaben und eines von drei Satzzeichen geschickt hat (Emoji seien der Einfachheit halber beiseite gelassen), sind es wahrscheinlich nur noch $(26 * 2 + 3)^2 = 55^2 = 3025$. Nun sind die meisten dieser Kombinationen keine richtigen Wörter, das heißt am Ende bleiben nur noch sehr wenige Möglichkeiten übrig, höchstwahrscheinlich eines der folgenden Wörter: “Ja”, “jo”, “ne”, “no”, “hi”, “ha” oder etwas ähnliches. Dies entspricht vermutlich weniger als 20 Möglichkeiten, was der Inhalt der Nachricht gewesen sein könnte.

Weiteres Szenario: Alice beginnt eine Konversation mit zwei Zeichen, Bob antwortet mit zwei Zeichen. Nun ist die Wahrscheinlichkeit extrem hoch, das siche beide Gesprächspartner mit einem “hi” begrüßt haben.

Wie verschleiert 1984not nun die Länge Ihrer Nachrichten? 1984not verlängert alle Nachrichten auf eine Länge von mindestens 255 Zeichen. Für Nachrichten länger als 255 Zeichen wird die Länge auf das nächste Vielfache von 255 Zeichen aufgerundet. Dies geschieht mithilfe von PKCS7-Padding[1].

2.1.6 PKCS7-Padding

Bei PKCS7-Padding werden an das Ende einer Nachricht Bytes angehängt, die die Länge des Paddings als Wert enthalten. Als Formel mit Länge der Nachricht l und Länge des Paddings p:

$$p = 255 - (l \mod 255)$$

Beispiele:

- “`Hi!`” → `‘H’ ‘i’ ‘!’ 252 252 252 ...252 252 252`
- “`Hi Alice!`” → `‘H’ ‘i’ ‘ ’ ‘A’ ‘l’ ‘i’ ‘c’ ‘e’ ‘!’ 246 246 246 ...246 246`

2.1.7 Prekeys

Um ein neues Double-Ratchet zu beginnen muss derjenige, der das Gespräch beginnt, einen Master-Key herleiten. Dies erfordert neben dem öffentlichen Identitäts-Schlüssel des Gesprächspartners jedoch auch einen öffentlichen Kurzzeitschlüssel.

Es ist bei asynchroner Kommunikation nicht möglich, diesen Kurzzeitschlüssel zu Beginn des Gespräches von dem Gesprächspartner anzufordern, da dieser eventuell gar nicht mit dem Internet verbunden ist. Dementsprechend müssen Kurzzeit-Schlüssel vorgeneriert werden, sogenannte Prekeys.

Beim ersten Anmelden am 1984not Server erstellt jeder Nutzer eine Liste von Prekeys. Möchte nun Alice ein neues Double-Ratchet mit Bob beginnen, so holt sich Alice seine Prekeys und wählt zufällig einen davon aus. Dies geschieht mit `libsodiums` Funktion `randombytes_uniform` um Zufallszahlen ohne Bias zu erhalten.

2.1.8 Schlüssel-Management

Jeder Nutzer von 1984not hat die folgenden Schlüsselpaare:

- Ein Signatur-Schlüsselpaar (Ed25519) Der öffentliche Signatur-Schlüssel wird genutzt um den Nutzer eindeutig zu identifizieren.
- Ein Identitäts-Schlüsselpaar (Curve25519) Dieses Schlüsselpaar geht als Identitäts-Schlüssel in die Herleitung des Double-Ratchet Master-Schlüssels ein.
- 100 Prekey-Schlüsselpaare (Curve25519) Prekeys werden nach jeder Verwendung ausgetauscht und auch nach einer gewissen Zeit durchrotiert.

In den Kontaktdaten eines Nutzers werden der öffentliche Identitäts-Schlüssel sowie die öffentlichen Prekeys abgespeichert, zusammen mit einem Ablaufdatum und einer Signatur vom Signaturschlüssel.

Auf diese Weise kann die Echtheit der Prekeys und des öffentlichen Identitäts-Schlüssels überprüft werden und eine Replay-Attacke wird verhindert, indem alte Zeitstempel nicht akzeptiert werden.

2.1.9 Überprüfung des Gesprächspartners

Um sicherzustellen, dass der andere Gesprächspartner wirklich der ist, für den er sich ausgibt, muss der öffentliche Schlüssel manuell verifiziert werden. Dies geschieht über das Scannen eines QR-Code, in dem der öffentliche Signatur-Schlüssel kodiert ist.

2.1.10 aXoChat und das „Socialist Millionairs Protocol“

Das zentrale Problem jeder Public-Key-Infrastruktur besteht darin, für alle Teilnehmer die Echtheit der von ihnen bezogenen öffentlichen Schlüssel sicherzustellen. Kann das nicht gewährleistet werden, wird typischerweise das gesamte System durch Man-in-the-Middle-Angriffe verwundbar. Übliche Herangehensweisen sind der Einsatz zentraler Zertifizierungsstellen (SSL), verteilte Zertifizierungsmodelle (PGP-Web-of-Trust) oder der direkte Abgleich von Schlüsselabdrücken (Fingerprints). Ein von Boudot et al. entwickeltes Zero-KnowledgeProtokoll ermöglicht den geheimen und sicheren Vergleich zweier Abdrücke über einen ungesicherten Kanal. Dabei benötigen die Kommunikationspartner nur einmalig ein vergleichsweise schwaches Shared-Secret. Umsetzung: Folgende Aufgabe ist bekannt als Yaos Millionärsproblem[14]: Zwei Millionäre wünschen zu bestimmen, wer der reichere ist, ohne dabei ihre Kontostände offenzulegen. Eine Abwandlung davon ist das sogenannte Socialist Millionaires' Problem[13], bei dem die beiden lediglich daran interessiert sind, ob ihr Reichtum gleich ist. Formal ist in beiden Fällen eine Möglichkeit gefragt, wie Alice und Bob einen Funktionswert f(x,y,. . .) bestimmen können, ohne daß einer von ihnen zusätzliche Informationen über die Eingabewerte (x,y,. . .) erhält. Ein Verfahren mit dieser Eigenschaft wird als Zero-Knowledge-Protokoll bezeichnet. Ist das Ergebnis von f ein Wahrheitswert, so handelt es sich um einen Zero-Knowledge-Beweis. Weiterhin nennt man ein Protokoll fair, wenn keiner der Teilnehmer es mit nennenswertem Informationsvorsprung beenden kann. Ein Verfahren zur Lösung des Problems wurde von Borisov et al. unter der Bezeichnung “Socialist Millionaires' Protocol” (SMP) für das Programm Off-theRecord implementiert[11][10]. Off-the-Record ermöglicht die verschlüsselte Kommunikation über Instant-Messaging-Dienste wie in dem von 1984not aXoChat. Dabei bietet sich das hier vorgestellte Verfahren insbesondere an, weil die Kommunikationspartner häufig miteinander vertraut sind aber in der Regel über keinen bestehenden Kanal zur sicheren Schlüsselverifizierung verfügen. Eine genaue Beschreibung des Protokolls ist unter der Veröffentlichung von Sven Moritz Hallberg[12] zu finden.

2.2 Finden von Kontakten

In aXoChat können die Kontakte über einen eindeutigen Benutzernamen, Handy-Nummer und die Email-Adresse gefunden werden. Bei der Installation von aXoChat kann jeder Benutzer entscheiden, wie er in aXoChat gefunden werden möchte. Es besteht die Möglichkeit komplett anonym zu bleiben und nur über den Benutzernamen gefunden zu werden. Des Weiteren kann man, wenn gewollt, über die Handy-Nummer oder die Email-Adresse ebenfalls gefunden werden. Für das Anmelden auf unseren User-Management-Servern muss bei der Installation entweder eine Handy-Nummer oder eine E-Mail Adresse angegeben werden. Jeder neue Anwender von aXoChat wird dann per SMS-Nachricht oder per E-Mail von 1984not Security GmbH authentifiziert. Nach dem erfolgreichen Authentifizieren kann nun jeder Anwender jeder Zeit in aXoChat entscheiden, wie er auf unseren User-Management-Servern gefunden werden möchte. Wir als 1984not Security GmbH erzeugen für jeden neuen Anwender auf unseren User-Management-Servern einen Account, der nur über die erzeugten Hash-Keys aus dem angegeben Benutzernamen, Handy-Nummer und E-Mail Adresse besteht. Das bedeute dass wir als 1984not Security GmbH in keinem Fall auf den originalen Benutzernamen, Hand-Nummer und E-Mail Adresse zugreifen können. Jeder Benutzer ist auf unseren User-Management-Servern anonym gespeichert. Hierdurch ist es gewährleistet das eventueller Datendiebstahl auf unseren Servern keine Möglichkeit bietet, hier Benutzerdaten zu lesen oder auch zu manipulieren. 1984not Security GmbH betreibt die User-Management-Servern nur in Deutschland was zusätzliche Sicherheit bedeutet. Keine Behörde oder Institution kann über uns Einblick auf die gespeicherten Nutzerdaten erhalten, da wir nur Hash-Keys der Anwender von aXoChat speichern. Die verwendete Technologie hierzu von 1984not Security GmbH macht dies einfach unmöglich.

2.3 Dateiversand

Über die App verschickte Dateien wie z.B. Bilder werden lokal mit der `blobcrypt`-Bibliothek[4] verschlüsselt und authentifiziert. Blobcrypt zerlegt die Datei hierbei in einzelne, mit der ChaCha20-Poly1305-Konstruktion verschlüsselte und authentifizierte Blöcke. Details dazu auf der Webseite der Bibliothek.[4]

Gleichzeitig wird der schlüssellose Blake2B-Hash der Datei ermittelt. Dies geschieht mithilfe der Funktion `crypto_generichash` von `libsodium`.

Der Schlüssel zum verschlüsseln wird hierbei mit `libsodiums` `randombytes_buf` erzeugt.

Anschließend wird die verschlüsselte Datei per HTTPS auf den 1984not Fileserver hochgeladen, wo sie als Dateinamen ihren Blake2B-Hash erhält. Hierbei ist der Hash die einzige Zugangsbeschränkung. Ein Angreifer kann eine Datei nur finden, wenn der Hash der Datei bekannt ist. Sollte es einen Sidechannel im Fileserver geben, der das herausfinden der Dateinamen erlaubt, so schützt `blobcrypt` immer noch adäquat.

Der symmetrische Schlüssel zum entschlüsseln der Datei und die URL der hochgeladenen Datei werden in einer regulären Nachricht per Double-Ratchet an den Empfänger gesendet. Auf der Empfangsseite muss nun die Datei vom Fileserver geladen und entschlüsselt werden.

2.4 Lokaler Speicher auf Android

Für das Abspeichern von Benutzerdaten und Chats werden die nachfolgend beschriebenen Mechanismen von Android und anderen Anbietern verwendet. Keystore In dem Keystore von Android werden von aXoChat alle sicherheitsrelevanten Daten gespeichert. Bei diesen Daten handelt es sich um die Passwörter und die Zugriffsdaten auf unsere 1984not Security Server. Das Android Keystore System schützt das Schlüsselmaterial von aXoChat vor unbefugter Nutzung. Erstens verhindert Android Keystore die unbefugte Nutzung von Schlüsselmaterial außerhalb des Android-Geräts, indem es die Extraktion des Schlüsselmaterials aus Anwendungsprozessen und aus dem Android-Gerät als Ganzes verhindert. Zweitens hindert Android KeyStore die unbefugte Nutzung von Schlüsselmaterial auf

dem Android-Gerät, indem Apps (hier unsere aXoChat App) die autorisierte Verwendung unserer Schlüssel angeben und diese Einschränkungen dann außerhalb der Prozesse der Apps durchsetzen.

2.4.1 SQLCipher

Die SQLCipher Datenbank verwenden wir zum Verschlüssen des Chatverlaufs mit Chatinhalt sowie für die aXoChat Adressenliste. SQLCipher ist eine Open-Source-Bibliothek, die eine transparente, sichere 256-Bit-AES-Verschlüsselung von SQLite-Datenbankdateien bietet. SQLCipher wurde von vielen kommerziellen und Open-Source-Produkten als sichere Datenbanklösung eingesetzt und ist damit eine der beliebtesten verschlüsselten Datenbankplattformen für mobile, eingebettete und Desktop-Anwendungen.

2.4.2 Shared Preferences

Für Einstellungen der aXoChat App wird die Technologie Shared Preferences von dem Android Betriebssystem verwendet. Diese Einstellungen sind nicht Sicherheitsrelevant. Hierbei handelt es sich um z.B. die Farbeinstellungen, Hintergrund oder Informationen über Klingeltöne allgemein. Diese Shared Preferences sind als Schlüsselpaare abgespeichert. In der aXoChat App werden aber trotz nicht Sicherheitsrelevanter Daten die Schlüsselpaare einfach verschlüsselt. Hierzu wird die von Android zur Verfügung gestellte SecretKeyFactory mit der Einstellung „PBKDF2WithHmacSHA1“ verwendet.

2.5 XMPP-Server

Das Extensible Messaging and Presence Protocol (XMPP, englisch für erweiterbares Nachrichten- und Anwesenheitsprotokoll; früher Jabber,[10] engl. “(daher-)plappern”) ist ein offener Standard eines Kommunikationsprotokolles, welches von der Internet Engineering Task Force (IETF) als RFC 6120, 6121 und 6122 veröffentlicht wurde. XMPP basiert auf dem XML-Standard und ermöglicht den Austausch von Daten. Es wird für das Instant Messaging von aXoChat eingesetzt. XMPP und seine Erweiterungen unterstützen Funktionen zur Nachrichtenübermittlung, Multi-User Chat, also Gruppen-Chats mit mehreren Benutzern, Anzeigen des Online-Status, Versendung von digitalen Zertifikaten und viele weitere Dienste. Die Netz-Architektur erinnert dabei an das Simple Mail Transfer Protocol (SMTP). Nachrichten werden von aXoChat zum 1984not Security eigenen Server, von dort dann zum Empfänger (aXoChat) weitergeleitet. Auch sind isolierte Netzwerke, beispielsweise in Firmen-Intranets möglich. Jeder Benutzer ist auf unseren XMPP-Servern anonym über eine Hash-Key gespeichert. Hierdurch ist es gewährleistet, das eventueller Datendiebstahl auf unseren XMPP-Servern keine Möglichkeit bietet hier Benutzerdaten zu lesen oder auch zu manipulieren. 1984not Security GmbH betreibt die XMPP-Servern nur in Deutschland was zusätzliche Sicherheit bedeutet. Keine Behörde oder Institution kann über uns Einblick auf die gespeicherten Nutzerdaten erhalten, da wir nur Hash-Keys der Anwender von aXoChat speichern. Chats werden nur so lange auf dem XMPP-Server zwischengespeichert bis der Empfänger wieder online ist. Danach wird die Chat-Message unwiderruflich gelöscht. Die verwendete Technologie hierzu von 1984not Security GmbH macht dies einfach unmöglich.

2.6 Sonstiges

2.6.1 Quellcode der Implementierung des Double-Ratchet

Unsere Implementierung des Double-Ratchet ist OpenSource (Freie Software) und der Quellcode kann auf GitHub eingesehen werden[5].

2.6.2 Implementierung der Krypto-Primitive

Alle in der Implementierung des Double-Ratchet verwendeten Krypto-Primitive stammen aus der Bibliothek `libsodium`. Details zu den verwendeten Funktionen können in der Dokumentation von `libsodium` nachgelesen werden[3].

2.6.3 Überschreiben von Schlüsseln im Arbeitsspeicher

Alle in der Implementierung des Double Ratchet verwendeten Schlüssel werden nach Verwendung im Speicher mit nullen überschrieben. Hierbei kam `libsodiums` Funktion `sodium_memzero` zum Einsatz um sicherzustellen dass entsprechende Schreibzugriffe nicht vom Compiler wegoptimiert werden.

2.6.4 Schutz von Schlüsseln im Arbeitsspeicher

In der Implementierung des Double Ratchet wurden besonders wichtige Schlüssel im Speicher durch Guard-Pages und Cookie-Values vor und nach den entsprechenden Speicherbereichen geschützt. Ausserdem wird die Auslagerung auf Festspeicher (swapping) dieser sensitiven Informationen verhindert. Zum Einsatz kamen hier `libsodiums` Funktionen `sodium_malloc` und `sodium_free`. Nach Verwendung werden diese Speicherbereiche zusätzlich gesperrt, sodass jeder Zugriffsversuch einen Absturz des Programmes zur Folge hat, womit ein Angriff weiter erschwert wird. Für diesen Zweck wird die Funktion `sodium_mprotect_noaccess` verwendet.

2.6.5 Zufallszahlen

Als Queelle für Zufallszahlen werden `libsodiums` `randombytes_buf` und `randombytes_uniform` verwendet. Diese verwenden auf Android wenn möglich den Linux System-Call `getrandom` und falls nicht verfügbar (auf alten Linux-Kerneln) die Datei `/dev/urandom`.

2.6.6 Erstellen des Identitäts-Schlüsselpaars

Bei der ersten Nutzung der 1984not-App wird der Nutzer aufgefordert über den Bildschirm zu streichen um zusätzlichen Zufall für das Generieren eines asymmetrischen Schlüsselpaares zu erzeugen.

Die Rohdaten aus den Fingerbewegungen werden mithilfe einer Password-Hash-Funktion zu einem Hash umgewandelt. Dieser Hash wird per bitweise exklusivem Oder mit einer vom Betriebssystem zur Verfügung gestellten Zufallszahl kombiniert. Hauptquelle von Zufall ist hierbei der Zufallszahlengenerator des Betriebssystems, die Fingerbewegungen sind nur eine zusätzliche Sicherheitsmassnahme falls das Gerät nur Zufallszahlen mit geringer Entropie erzeugt.

$$\texttt{seed} = \texttt{os\_random} \oplus \texttt{pwhash}(\texttt{finberbewegungen})$$

Als Paswort-Hash kommt hierbei `libsodiums` `crypto_pwhash` Funktion mit den Parametern `crypto_pwhash_OPSLIMIT_INTERACTIVE`, `crypto_pwhash_MEMLIMIT_INTERACTIVE`, `crypto_pwhash_ALG_DEFAULT` zum Einsatz. Hierbei handelt es sich um die Funktion `argon2i v1.3` mit dem Speicherverbrauchs-Parameter 33554432 und dem Rechenzeit-Parameter 4.

Das Resultat aus diesem exklusivem Oder fungiert als zwei Seeds für ein Ed25519-Schlüsselpaar und ein Curve25519-Schlüsselpaar. Das Curve25519-Schlüsselpaar wird eingesetzt um den Master-Key des Double-Ratchet abzuleiten, während das Ed25519 verwendet wird um das Curve25519-Schlüsselpaar und die Prekey-Liste zu signieren.

Aus dem Seed abgeleitet werden die Schlüsselpaare mit `libsodiums` `crypto_sign_seed_keypair` und `crypto_box_seed_keypair` abgeleitet.

Literatur

[1] Section 6.3, RFC 5652

[2] Double-Ratchet Spezifikation `https://whispersystems.org/docs/specifications/doubleratchet/`

[3] `libsodium` Krypto-Bibliothek `https://download.libsodium.org/doc/`

[4] `blobcrypt` Bibliothek zum Verschlüsseln von Dateien mit libsodium `https://github.com/jedisct1/blobcrypt`

[5] `molch`, 1984not's Implementierung des Double-Ratchet `https://github.com/1984not-GmbH/molch`

[6] Email-Icon, Lizenz: Public Domain `https://sourceforge.net/projects/openiconlibrary/`

[7] Switch-Icon, Server-Icon, Schloss-Icon, Smartphone-Icon, Autor: OSA, Lizenz: Creative-Commons Attribution-ShareAlike (CC-BY-SA 3.0) `http://www.opensecurityarchitecture.org/cms/library/icon-library`

[8] Google Protocol Buffers `https://developers.google.com/protocol-buffers/`

[9] Protbuf-C Bibliothek für Google Protocol Buffers `https://github.com/protobuf-c/protobuf-c`

[10] Borisov, Nikita, Ian Goldberg und Eric Brewer: O-the-record communication, or, why not to use PGP. In: WPES '04: Proceedings of the 2004 ACM workshop on Privacy in the electronic society, Seiten 77–84, New York, NY, USA, 2004. ACM.

[11] Off-the-Record Messaging Protocol version 2. `http://www.cypherpunks.ca/otr/Protocol-v2-3.1.0.html`

[12] Individuelle Schlüsselverifikation via Socialist Millionaires' Protocol Sven Moritz Hallberg `sm@khjk.org` Juni 2008

[13] Jakobsson, Markus und Moti Yung: Proving Without Knowing: On Oblivious, Agnostic and Blindfolded Provers. Lecture Notes in Computer Science, 1109:186–200, 1996

[14] Yao, Andrew: Protocols for secure computations. In: Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science, Seiten 160–164, 1982