Kryptographie nachrechnen am Beispiel von Threema

Martin Christian

12.03.2022

Motivation

- Nutzung von Threema
- Interesse an kryptographischen Verfahren
- Berufliche Erfahrung durch Zertifizierungsverfahren

Überblick

Einstieg
- Motivation
- Überblick

Setup

Verfahren
- Curve25519
- XSalsa20

Nachrechnen
- Threema Backup
- Nachrichten
- Missing

ENDE
- Bibliographie
- Kontakt

Software

- *Threema* ist ein *Messenger mit Fokus auf Sicherheit und Privatsphäre* (`https://threema.ch/de`)
- Threema nutzt die Kryptobibliothek *NaCl* (`https://nacl.cr.yp.to`)
- *openMittsu* ist eine inoffizielle Desktop-Version (`https://www.openmittsu.de`)
- openMittsu nutzt die Kryptobibliothek *Sodium* (`https://doc.libsodium.org`), ein aktiv entwickelter Fork von NaCl

Bernstein-Lange-Universum

- Daniel Julius Bernstein: `https://cr.yp.to/`
- Tanja Lange: `http://hyperelliptic.org/`
- Universum: `https://nacl.cr.yp.to/`
  - Elliptic Curve *25519*: Diffie-Hellman Schlüsselaustausch
  - Blockchiffre *Salsa20*
  - Message Authentication Code *Poly1305*

Threema

1. Threema-App installieren
2. Threema ID erzeugen
3. Backup der Threema ID erstellen und exportieren

openMittsu

1. Threema ID aus Backup importieren
2. Patch von openMittsu zur Ausgabe von ein- und ausgehenden Nachrichten
3. Nachrichten senden und empfangen

Kein Sagemath

```
 Aktionen  Rückgängig  Paket  Auflöser  Suchen  Optionen  Ansichten  Hilfe
C-T: Menü  ?: Hilfe  q: Beenden  u: Update  g: Vorschau/Herunterladen/Installieren/Entferne Pakete
aptitude 0.8.12 @                                       Datenträger: +2.         DL: 404 MB
pi   sagemath                                           +120 MB   <keine>   9.0-1ubuntu4
piA  sagemath-common                                    +205 MB   <keine>   9.0-1ubuntu4
piA  sagemath-database-conway-polynomials               +938 kB   <keine>   0.5-7
piA  sagemath-database-elliptic-curves                  +10,6 MB  <keine>   0.8.1-4
piA  sagemath-database-graphs                           +3.600 kB <keine>   20161026+dfsg-
piA  sagemath-database-mutually-combinatorial-designs   +43,0 kB  <keine>   20140630-5
piA  sagemath-database-polytopes                        +293 kB   <keine>   20170220-4
piA  sagemath-doc                                       +1.518 MB <keine>   9.0-1ubuntu4
piA  sagemath-jupyter                                   +110 kB   <keine>   9.0-1ubuntu4
p    sc                                                           <keine>   7.16-4ubuntu3
p    scalapack-mpi-test                                           <keine>   2.1.0-2build1
p    scilab                                                       <keine>   6.1.0+dfsg1-1u
p    scilab-cli                                                   <keine>   6.1.0+dfsg1-1u
p    scilab-data                                                  <keine>   6.1.0+dfsg1-1u
p    scilab-doc                                                   <keine>   6.1.0+dfsg1-1u
p    scilab-full-bin                                              <keine>   6.1.0+dfsg1-1u
p    scilab-include                                               <keine>   6.1.0+dfsg1-1u
p    scilab-minimal-bin                                           <keine>   6.1.0+dfsg1-1u
p    scilab-test                                                  <keine>   6.1.0+dfsg1-1u
p    scotch                                                       <keine>   6.0.9-1
p    sdpa                                                         <keine>   7.3.14+dfsg-1
p    sdpam                                                        <keine>   7.3.14+dfsg-1
p    sdpb                                                         <keine>   1.0-3build6
piA  singular                                           +23,6 kB  <keine>   1:4.1.1-p2+ds-
Open Source Mathematical Software
```

Python

1. Pari-Bibliothek (`https://pari.math.u-bordeaux.fr`)
2. Python Interface for Pari: CyPari2
3. Eigene Tools: `github.com/christianix/cryptotools`

Definition

- $y^2 = x^3 + 486662 \cdot x^2 + x$
- Basispunkt $P_0 = (9, \sqrt{39420360})$, weil $9^3 + 486662 \cdot 9^2 + 9 = 39420360$

```
pari = cypari2.Pari()
E = pari.ellinit([0,486662,0,1,0], 2**255-19)
P = [pari.Mod(9, 2**255-19),
        pari.sqrt(pari.Mod(39420360, 2**255-19))]
```

Clamping

- Abbildung einer Zufallszahl auf einen privaten Schlüssel
- Siehe auch: *What's the Curve25519 clamping all about?* von Neil Madden

---

```
def clamp(n):
  """Curve25519 clamping by Matthew Dempsky"""

  n &= ~7
  n &= ~(128 << 8 * 31)
  n |= 64 << 8 * 31
  return n
```

Schlüsselpaare

- Randbedingung: Elliptische Kurve E und Basispunkt P
- Geheimer Schlüssel: Skalarer Zufallswert s
- Öffentliche Schlüssel: $\mathtt{PK} = s \cdot P$
- Unterschied zu Brainpool: Clamping und die Verwendung des X-Werts

---

```
s = int.from_bytes( sk_bytes, "little" )
sk = clamp(s)
pk_p = pari.ellmul(E, P, sk)
pk_i = gen_to_integer(pari.lift(pk_p[0]))
pk = pk_i.to_bytes(32, byteorder='little')
```

ECDH Schlüsselableitung

- Mein geheimer Schlüssel s_m (Skalar)
- Ihr öffentliche Schlüssel $\mathtt{PK}_o$ (X-Koordinate)
- *My* gemeinsamer Schlüssel: $\mathtt{K}_m = s_m \cdot \mathtt{PK}_o = s_m \cdot s_o \cdot P$
- *Other* gemeinsamer Schlüssel: $\mathtt{K}_o = s_o \cdot \mathtt{PK}_m = s_o \cdot s_m \cdot P$

```python
def curve25519_pk_mult(pari, s, x):
  E = pari.ellinit([0,486662,0,1,0], 2**255-19)
  x_elem = pari.Mod(x, 2**255-19)
  # y = Sqrt(x^3 + 486662 * x^2 + x)
  y = x**3 + 486662 * x**2 + x
  y_elem = pari.sqrt(pari.Mod(y, 2**255-19))
  P = [x_elem, y_elem]
  return pari.ellmul(E, P, s)
```

Salsa-Familie

- Zustand $s = (s0, \ldots, s15)$ besteht aus $16 \cdot 32$ Bit Wörtern
- Zustand s wird initialisiert mit Konstante, Nonce und Schlüssel
- Kern ist *Mixer* mit r Runden: $s = \texttt{doubleround}^{\frac{r}{2}}(s)$
- Schlüssel ist immer 256 Bit lang
- Erzeugt einen Schlüsselstrom k der Länge $|k|$
- Verschlüsselung: $c = k' \oplus m$, mit $|k'| = |m| \leq |k|$

Gegenüberstellung

Name	Nonce-Länge	Finalisierung	Ausgabelänge
HSalsa	128 Bit	$\texttt{select}(s)$	256 Bit
Salsa	64 Bit	$s^{INIT} + s^{FIN}$	512 Bit
XSalsa	192 Bit	$\text{Salsa}(k', n_2, c)$	512 Bit

Anmerkungen:

- Bei (X)Salsa kommt ein 64-Bit Zähler als Eingabe hinzu
- XSalsa-Nonce: $|n_1| + |n_2| = 128 + 64 = 192$
- XSalsa Schlüsselabteilung: $k' = HSalsa(k, n_1)$

Python-Tools

`salsa.py` Implementierung von HSalsa, Salsa und XSalsa

`xsalsa-decrypt.py` Skript zur Entschlüsselung von XSalsa

`salsa-test.py` Testvektoren zur Salsa-Implementierung

Beispiel

```
# Test 6: Chapter 10
tvector.clear()

tvector["nonce"] = bytearray(16)
tvector["key"] = bytearray.fromhex("4A5D9D5B...")
k1 = hsalsa(**tvector)
tvector.clear()
tvector["nonce"] = bytearray.fromhex("69696ee9...")
tvector["counter"] = 0
tvector["key"] = k1
expected = bytearray.fromhex("eea6a725...")
run_test(xsalsa, '6 - Chaper 10', tvector, expected)
```

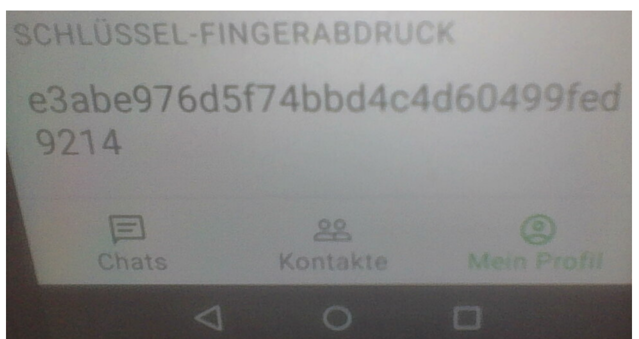
Vorgehen

1. Backup entpacken (ZIP)
2. Backup des Schlüssels in Binärform wandeln (Base32)
3. Salt abtrennen
4. Schlüssel mit Salt und Passwort berechnen (PBKDF2)
5. Backup entschlüsseln (XSalsa20)
6. Prüfe Korrektheit der Entschlüsselung (SHA2-256)

(Basierend auf *Cryptography Whitepaper* von Threema.)

Demo

Nachtrag



1. Hash des PK: `SHA256(my_pk.bin)`
2. Die ersten 16 Byte davon sind der Fingerprint

Vorgehen

1. openMittsu patchen
2. ECDH Schlüsselableitung: $\mathtt{DHS} = \mathtt{s}_m \cdot \mathtt{PK}_o$
3. Salsa-Hash: $k = \mathtt{HSalsa20}(\mathtt{DHS})$
4. Nachricht entschlüsseln

openMittsu Patch

```
--- a/src/network/ProtocolClient.cpp
+++ b/src/network/ProtocolClient.cpp
@@ -737,7 +740,10 @@ namespace openmittsu {
  LOGGER_DEBUG("Sending Message to Contact {} with ID {}.",
    contactMessage->getMessageHeader().getReceiver().toString(),
    contactMessage->getMessageHeader().getMessageId().toString());

  openmittsu::messages::MessageWithPayload messageWithPayload(contactMessage->getMessageHeader(),
    contactMessage->getContactMessageContent()->toPacketPayload());
+ std::cout << "Plain message: " << QString(
    messageWithPayload.getPayload().toHex()).toStdString() << std::endl;
  openmittsu::messages::MessageWithEncryptedPayload messageWithEncryptedPayload(
    messageWithPayload.encrypt(m_cryptoBox));
+ std::cout << "Encrypted message: " << QString(
    messageWithEncryptedPayload.getEncryptedPayload().toHex()).toStdString() << std::endl;
+ std::cout << "Nonce: " << QString(
    messageWithEncryptedPayload.getNonce().getNonce().toHex()).toStdString() << std::endl;

encryptAndSendDataPacketToServer(messageWithEncryptedPayload.toPacket());
```

Gemeinsamer Schlüssel

```
# Calculate Diffie-Hellman shared key
dh_p = curve25519_pk_mult(pari, my_sk, other_pk)
dh_i = gen_to_integer(pari.lift(dh_p[0]))
dhkey = dh_i.to_bytes(32, byteorder='little')

# Build input of salsa
kdfinput = {}
kdfinput["nonce"] = bytearray(16)
kdfinput["key"] = dhkey

hsalsa = HSalsa20()
k = hsalsa(**kdfinput)
```

Demo

Was fehlt noch?

- Integritätsschutz *Poly1305*
- Client-Server Kommunikation

- Cryptography Whitepaper. Threema. 21.6.2021.
- Cryptography in NaCl. Daniel J. Bernstein. 10.3.2009.
- What's the Curve25519 clamping all about? Neil Madden. 28.05.2020.

Kontakt

- Martin Christian
- Mail: martin at christianix.de