# Formal verification of Sesame

Emmanuel MERA

June 30, 2025

## 1 Introduction

### 1.1 Internship objectives

### 1.2 State of the art

### 1.3 Motivations

### 1.4 Overview of the work done

## 2 The protocol behind Signal

### 2.1 Signal

Signal is an open source application allowing users to communicate between them securely. It uses a similar protocol as in WhatsApp and is used by millions of active users. Each user may have several devices, adding and removing them whenever he pleases, that are kept synchronized. The conversations are encrypted end-to-end and the messages are transmitted from a device to another through a server.

### 2.2 X3DH

X3DH - or triple Diffie-Helman - is the key exchange protocol used by signal. It allows a device $A$ to initiate a secure and authenticated communication channel with another device $B$. When $A$ wants to initiate a conversation with $B$, the protocol goes as follow. On the server is present 3 cryptographic informations about $B$ :

- a public identity key $\mathrm{IK}_B^p$
- a public signature key $\mathrm{SK}_B^p$
- a public long time signed prekey $\mathrm{SPK}_B^p$, alongside a signature $\mathrm{sSPK}_B^p$
- a set, possibly empty, of public one-time per-key $\mathrm{OPK}_B^p$

$A$ fetches from the server a prekey bundle for $B$, which contains $\mathrm{IK}_B^p$, $\mathrm{SPK}_B^p$ and, when one is available, $\mathrm{OPK}_B^p$. The server then removes $\mathrm{OPK}_B^p$ from the set of one-time pre-key. $A$ first check the signed prekey against the signature using $\mathrm{SK}_B^p$. If the check fails, $A$ does not initiate any connection and stops. Else, $A$ generates an ephemeral key pair $\mathrm{EK}_A^s$ and $\mathrm{EK}_A^p$. It then computes the Diffie-Hellman keys :

$$\begin{aligned} g1 &= DH(\mathrm{SPK}_B^p, \mathrm{IK}_A^s) \\ g2 &= DH(\mathrm{IK}_B^p, \mathrm{EK}_A^s) \\ g3 &= DH(\mathrm{SPK}_B^p, \mathrm{EK}_A^s) \\ g4 &= DH(\mathrm{OPK}_B^p, \mathrm{EK}_A^s) \end{aligned}$$

$A$ computes the last derivation only when some $\mathrm{OPK}_B^p$ is available. When those are computed, $A$ can derive the shared key :

$$K = H(g1||g2||g3||g4)$$

$H$ is a hash function (e.g. SHA2). In the case g4 is not available, it is not concatenated with the other and the derived key is $H(g1||g2||g3)$.

When $\mathrm{OPK}_B^p$ is not available, $g4$ is not added in the computation of the key. $DH$ designates a Diffie-Hellman function, $H$ a hash function and $||$ a concatenation function. $A$ can finally publish on the server $IK_A^p$ alongside a first message, encrypted with an AEAD, and the ephemeral key as additional data for authentication. The process $A$ can then terminate and use the derived key $PK$ in future communication. When $B$ fetch the initial message from the server, with the additional data $\mathrm{IK}_A^p$, it can locally compute the public key $PK$ and try de decipher the message. If the message is deciphered and $IK_A^p$ authenticated, the key derived is the correct one and the protocol terminates, returning the derived key.

### 2.3 Double-Ratchet

The double-ratchet protocol is used in signal to update the symmetric key at each message. Because the tool we used for the cryptographic model (CryptoVerif) does not allow for such manipulations of the key, we don't implement the double-ratchet protocol. Instead, we use the same key for all communications with this session.

### 2.4 Sesame

Sesame is the session management part of signal. It keeps the devices of a user synchronised and manages the communication between users. Its API allows a device to send messages encrypted to other users, ensuring that all of the devices of the recipients receives the message and all the other devices of the sender are kept up-to-date. It also allows to decrypt messages received from the server.

## 3 CryptoVerif

### 3.1 Overview of the tool

CryptoVerifis theorem prover for cryptography. It has the ability to write proofs of cryptographic protocols in the computational model. The proofs are derived using rewriting rules, transforming the game until the queries are proven. It is able to prove indistinguishability between two games and prove authentication queries.

### 3.2 Model

CryptoVerifis a tool made to prove small models. When given a too complex model (i.e. with a lot of branches and oracles), it becomes very difficult to write any proofs. Also, it does not provide any mechanism to manipulate a state. Therefore, we tried to model only the cryptographic part of the protocol using this tool, letting all the algorithmic part for FStar. The cryptographic part of Signal is in two parts :

1. The key derivation protocol
2. The encrypted exchanges using this key

We model those two parts in CryptoVerif, and prove them secure in a way that will be make precise later.

To write this model, we rely on several cryptographic primitives from the standard library of CryptoVerif:

**Diffie-Hellman** The Diffie-Hellman function serves in deriving the key. The documentation of Signal recommends to use Curve25519. In the standard library of CryptoVerif, there is a model of Curve25519. We add to this model the Gap Diffie-Hellman assumption. We write $G$ the group of Curve25519 and $Z$ the set of exponents. We write $g$ a generator of the group and $g^a$ for $a \in Z$ the exponentiation function.

**Hash** The hash function is used to finalize the key derivation. In CryptoVerif, we model with the Radom Oracle Model.

**Signature** The signature is used only to sign the long time prekey and produce $\mathrm{SPK}^s_*/p$. It satisfies the UF-CMA assumption. We write Sign the set of signatures, and note s$X$ the signature of $X$.

**AEAD** The AEAD is used to encrypt the messages after deriving the key. We assume that the AEAD is IND-CPA and INT-CTXT for the encrypted messages, with authentication for the additional data.

To describe the structure of the model, we will use the representation of 1 to represent the oracles OA ; (OB | OC).

```
Oracle OA
    Oracle OB
    Oracle OC
```

**Algorithm 1:** Graphic representation of OA ; (OB | OC)

The cryptographic model has a structure given by algorithm 2. The random oracle has been omitted for clarity but is included in the CryptoVerifmodel. The full model is given in **!!! GIVE REF FOR FULL CRYPTO MODEL !!!**.

The model that we have written should be read as followed :

At the beginning, both Alice and Bob calls Ogenprinc to generate identity keys that they upload to the server. Bob wants to accept incoming communications so compute a long-time signed prekey $\mathrm{SPK}^p_B$ with Osend1 and a bunch of $\mathrm{OPK}^p_B$ calling several times Osend2. All of those are then uploaded to the server. Alice wants to start communicating with Bob so gather Bob's keys from the server. Alice then generates the initial message using Orecv1 - or Orecv2 in case no one-time prekey were available on the server at the time - and upload the values to the server. Bob at some point polls this initial message from the server and calls Osend3 - or Osend4 in case Alice did not use any one-time prekey - to verify the gathered informations. Once Bob authenticated Alice, they can both encrypt and decrypt messages using the appropriate Oend and Odec.

## 3.3 Implementations choices

Some choices have been made to diverge from the documentation to have a simpler model. With the given model, CryptoVerifalready struggles to run the proofs.

```
(* Generates identities and returns $\mathrm{IK}^p_*$ and $\mathrm{SK}^p_*$.                *)
Oracle Ogenprinc: G * pkey
    (* Used to simulate the leaking of a key to the adversary              *)
    Oracle OCorrupt
(* Generates $\mathrm{SPK}^p_B$ alongside its signature $s\mathrm{SPK}^p_B$                     *)
Oracle Osend1 ($IK^p_b$: G): G*Sign
    (* --------------------------------------------------                  *)
    (* This fist part is for the case where we are playing the protocol with
      a one-time prekey                                                    *)
    (* --------------------------------------------------                  *)
    (* Generates $\mathrm{OPK}^p_B$, a one time prekey                                   *)
    Oracle Osend2: G
        (* Receives the initial message from $A$ and                       *)
        (* continues if the derived key is valid.                          *)
        Oracle Osend3 ($IK^p_A$: G, $SK^p_A$: pkey, $EK^p_A$: G, initial_message)
            (* Tries to decrypt the cipher.                                *)
            Oracle Odec1(cipher, nonce): message

            (* Tries to encrypt the cipher using the given nonce.          *)
            Oracle Oenc2(message, nonce): cipher
    (* --------------------------------------------------                  *)
    (* This second part is for the case where we are playing the protocol
      without a one-time prekey                                            *)
    (* --------------------------------------------------                  *)
    (* Receives the initial message from $A$ and                           *)
    (* continues if the derived key is valid.                              *)
    Oracle Osend4($IK^p_A$: G, $SK^p_A$: pkey, $EK^p_A$: G, initial_message)
        (* Tries to decrypt the cipher.                                    *)
        Oracle Odec3(cipher, nonce): message

        (* Tries to encrypt the cipher using the given nonce.              *)
        Oracle Oenc4(message, nonce): cipher
(* --------------------------------------------------                      *)
(* Creates an initial message if the prekey is well signed.                *)
(* --------------------------------------------------                      *)
Oracle Orecv1($IK^p_A$: G, $IK^p_B$: G, $SK^p_B$: pkey, $SPK^p_B$: G, $sSPK^p_B$, $OPK^p_B$: G)
    (* Tries to encrypt the cipher using the given nonce.                  *)
    Oracle Oenc1(message, nonce): cipher

    (* Tries to decrypt the cipher.                                        *)
    Oracle Odec2(cipher, nonce): message
(* --------------------------------------------------                      *)
(* Creates an initial message if the prekey is well signed.                *)
(* This version does not make use of a one-time prekey.                    *)
(* --------------------------------------------------                      *)
Oracle Orecv2
    (* Tries to encrypt the cipher using the given nonce.                  *)
    Oracle Oenc3(message, nonce): cipher

    (* Tries to decrypt the cipher.                                        *)
    Oracle Odec4(cipher, nonce): message
```

### 3.3.1 Adaptation to CryptoVerifAEAD hypothesis

The AEAD in CryptoVerifrequires that the nonce are used exactly once to encrypt messages for each key. The protocol is designed to create a common symmetric key. Therefore, there is a problem of distributing the nonce correctly between Alice and Bob. We saw two solutions available to us :

1. Adding a prefix to the nonces, for example adding a 1 for the nonce of Bob and a 0 for the nonce of Alice as a prefix.

2. Cutting the key in two parts, one part for Alice to encrypt, another for Bob to encrypt.

The first solution appeared to be the one that would keep us the closest to the original protocol. Unfortunately, it increases a lot the complexity of the model. Applying the hypothesis int_ctxt becomes impossible for CryptoVerifbecause of the usage of the same key in different oracle. CryptoVerifgenerates a great amount of branches to account for those usages making it impossible for the tool to later conclude the proof.

The second solution makes it possible for CryptoVerifto proofs the desired theorems. The key being cutted, the two parts can be assumes independently generated and makes the application of the int_ctxt hypothesis possible and does not create intricate connections between the orales.

### 3.3.2 The composition of the additional data

The data authenticated with the initial message are slightly different from the documentation. In our model, we authenticate $\mathrm{IK}_A^p$, $\mathrm{SK}_A^p$, $\mathrm{IK}_B^p$, $\mathrm{SK}_B^p$ and the ephemeral generated by $A$.

### 3.3.3 An empty initial message

In the protocol, the initial message is assumed to be secret and therefore can carry real data. In the model however, we do not prove the secrecy of the first message. This is because proving the secrecy of this message requires to add a branching at the very beginning of the protocol and essentially doubles or even more the computation time of the proof. We were not able to make the prover run using this branching so the first message is not authenticated. We replaced it with a public message which in the implementation is a zero-length bitstring.

### 3.3.4 A case for neutral keys

If Alice is given a set of keys which are all neutral elements, it derives a public key which is computable by the attacker (the neutral element). In theory, it should not be a problem because the probability of Bob generating a set of keys that are all the neutral element is negligable in the model and impossible in reality (because of the choice of exponents). But for CryptoVerifto understand that this is not a problem, it requires to handpick exactly what cryptographic assumption to apply and where, therefore making the proof less stable and more tedious to understand. It will though probably be done in the future.

## 3.4 Theorems proved

### 3.4.1 On the definition of corrupted session

The definition of a corrupted session is subtle. Indeed, even in the case of identity key corruption with OCorrupt, a session might not be corrupted. A session is honest (not corrupted) if one of the following is true :

- The identity key of the other correspondent is not corrupted, e.g. when $A$ receives an $\mathrm{IK}_B^p$ that has not been corrupted.

- The ephemeral prekey is valid (that is, generated by some oracle in the protocol and not by the adversary), e.g. the signed prekey $\mathrm{SPK}_B^p$ received by $A$ must have been generated by some $B$, or the ephemeral $\mathrm{EK}_A^p$ received by $B$ has been sent by some $A$.

### 3.4.2 Secrecy

Message secrecy sent using the different Oenc and Odec has been proven. We have proven the indistinguishability between the two following game :

- We encrypt all messages using the AEAD
- The messages we encrypt are not the original one but bitstring filled with zero of the same length when the session is not corrupted. If the session is corrupted, the messages are not secret so we do not replace the original message.

In other words, it says that in the case the sessions is not corrupted, the attacker does not make the difference between the original message encrypted and the zero of this message encrypted.

### 3.4.3 Authentication of initial message

The authentication of the initial message is done only when $A$ is not corrupted. Indeed, if $A$ is corrupted then the attacker can forge an initial message. We prove that for each initial message authenticated by $B$ has been sent by some $A$.

In the case where the protocol with a one-time prekey, this relation can also been proved to be injective, that is the relation that, for each authenticated initial message received by $B$ associate the event of $A$ sending the message, is injective.

When the protocol does not run with a one-time prekey, the attacker can replay the initial message of $A$ with the same keys and break the injectivity.

### 3.4.4 Authentication of other messages

Further messages are authenticated separately for two reasons. First it makes the proof goal easier because the authentication of the initial message can be proven very early in the proof. Second, it is not possible to prove the injectivity of the relation. Indeed, we do not prevent against decrypting two times the same message and therefore there is a replay attack against the injectivity.

## 3.5 Limitations

### 3.5.1 Of the model

### 3.5.2 Of the tool used

## 3.6 Future work

# 4 Sesame in F*

## 4.1 Overview of the tool

## 4.2 Client implementation

## 4.3 Network model

## 4.4 Server implementation

## 4.5 Interface with CV2FStar

# 5 Adjustments to the code extracted from CryptoVerif

## 5.1 Cryptographic functions

## 5.2 Missing proofs

# 6 Cryptographic proofs in F*

## 6.1 Authentication

## 6.2 Forward secrecy

## 6.3 Post-compromise security

# 7 Conclusion

## 7.1 Work done

## 7.2 Other paths currently explored

## 7.3 Future work