Module 1: Formal Verification of Security Protocols

Lab 2: An Anonymous Message Fetching Protocol

David Basin, Sofia Giampietro Felix Linker, and Xenia Hofmeier

basin@inf.ethz.ch, sofia.giampietro@inf.ethz.ch, flinker@inf.ethz.ch, xenia.hofmeier@inf.ethz.ch

Autumn Semester 2025

Welcome to the second lab session of the Information Security Lab! This session is part of the first module on *Formal Verification of Security Protocols*.

Overview

SecureDrop is a whistleblowing system developed by the Freedom of the Press Foundation (FPF) that allows *sources* to anonymously share confidential information with *journalists*. SecureDrop has been deployed by many reputable newspapers, for example, the New York Times, The Guardian, and The Washington Post.

Currently, the FPF is designing an end-to-end encrypted protocol intended to supersede the original system design. In this new protocol, sources and journalists can exchange messages via a message delivery server. The delivery server works similarly to a letterbox in which sources and journalists can leave messages for each other. As part of the re-design, the FPF has developed a protocol that allows sources and journalists to fetch their message at a message delivery server without the server learning who the message is from or who it is for. At the same time, the server verifies that messages only are shared with those who they were intended for.

In this assignment, you will model this message fetching protocol, formalize its security guarantees, find an attack, and fix the protocol. We provide some rules for the submission at the end of this document and recommend you read them carefully in advance.

Part I: Flawed Protocol

The protocol has two types of participants: clients and servers.[1] We assume that servers already received some number of messages that they now seek to distribute to the correct clients. We ignore how these messages were submitted to the server. On a high-level the protocol works as follows:

[1] We simplify the protocol for this assignment and do not distinguish between sources and journalists.

1. A client queries a server for a new message.
2. The server chooses one of its stored messages and generates a challenge for that message.

3. If the client can solve the challenge, the server releases the message to the client and deletes the message.

Message Format and Key Material

Each server controls a long-term public/private key pair used for asymmetric encryption. We assume a secure Public-Key Infrastructure in that clients know every server's authentic public key. Each client controls a long-term public/private Diffie-Hellman key pair used for authentication.

Messages stored at the server have three components. In the skeleton file, the rule `Submission` generates messages stored at servers.

- Confidential message content, which we call "secrets."
- An ephemeral, unique Diffie-Hellman public key $pk = g^{k_E}$.[2]
- A challenge value v. The challenge value is the intended recipient's Diffie-Hellman public key pk_R raised to the ephemeral private key k_E, i.e., $v = pk_R^{k_E}$.

[2] Note that the corresponding ephemeral Diffie-Hellman private key k_E is nowhere stored.

By the Decisional Diffie-Hellman assumption, one cannot learn to which client public key the challenge value corresponds. This hides the intended recipient to the server.

Protocol Flow

The challenge-response protocol is run over a secure channel. To establish the secure channel, clients sample a symmetric key k at the beginning of the protocol, asymmetrically encrypt k under the server's public key, and send the corresponding ciphertext to the server. All other protocol messages are symmetrically encrypted under that key k.

The challenge-response protocol is depicted in Figure 1. It starts with the client sending a request to the server, which then chooses one of its stored secrets x and associated challenge value v and ephemeral public key pk. The server then generates a random value r and challenge c, raises the challenge value v to r, and symmetrically encrypts c under that value. It sends the client the encrypted challenge and the ephemeral public key raised to r. If the client can, it will decrypt the challenge and respond with a hash of the challenge. When the server receives the correctly hashed challenge, it sends the client the previously chosen secret x. Whenever a server sends a secret to a client, it deletes the secret afterwards. Every secret can only be fetched once.

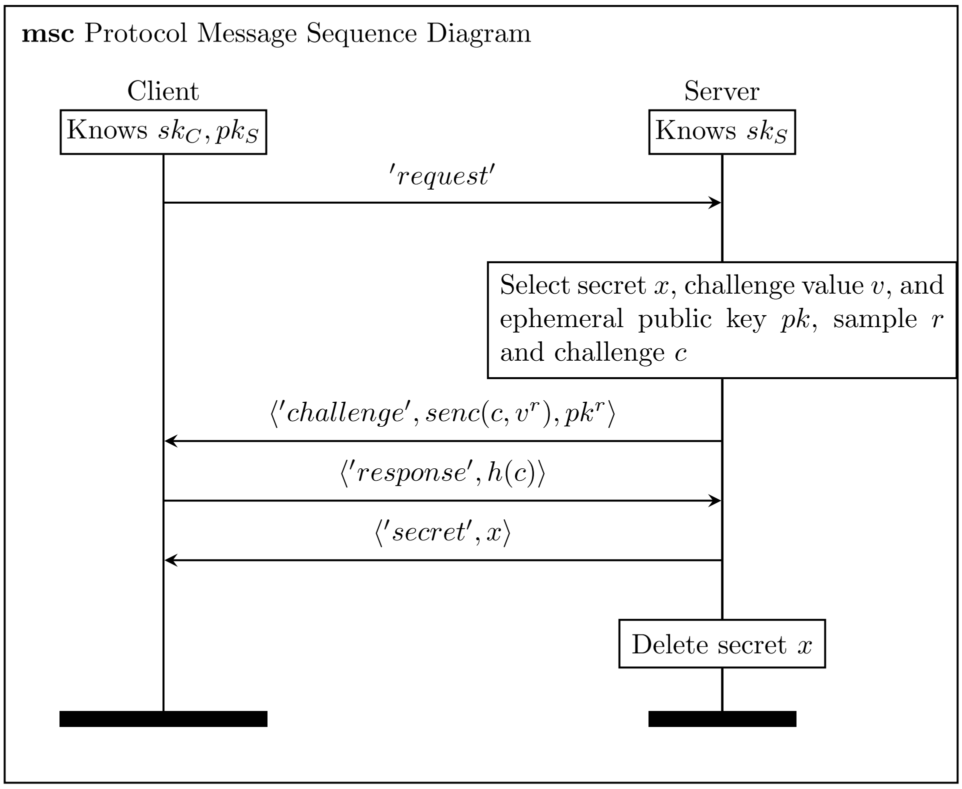
msc Protocol Message Sequence Diagram
Client
Server
Knows sk_C, pk_S
Knows sk_S
$'request'$
Select secret x, challenge value v, and ephemeral public key pk, sample r and challenge c
$\langle 'challenge', senc(c, v^r), pk^r \rangle$
$\langle 'response', h(c) \rangle$
$\langle 'secret', x \rangle$
Delete secret x

Figure 1: Challenge-response protocol flow. The secure channel is omitted. sk_C is the client private key, sk_S the server private key, and pk_S the corresponding server public key.

Action Facts

Your model must contain the following action facts, added to the appropriate rules.

- `ClientBegin(cl, s, k)`: The client `cl` starts the protocol with server `s`. It uses key `k` for the secure channel.
- `ServerChallenge(s, k, c, x)`: The server `s` receives a client request encrypted with `k`. It selects the secret `x` and samples the challenge `c`.
- `ClientRespond(cl, s, k, c)`: The client `cl` receives and decrypts challenge `c` from server `s`. The client uses key `k` for the secure channel.
- `ServerRelease(s, k, x)`: The server `s` receives a decrypted challenge from a client over a secure channel that uses key `k`. The server shares secret `x` with that client.
- `ClientReceive(cl, s, k, x)`: Client `cl` receives secret `x` from server `s`. The client uses key `k` for the secure channel.

Task

Model the challenge-response protocol (including the secure channel) as described above. Annotate all corresponding rules with the action facts from above. Our source skeleton file already provides you with rules for client and server key generation, and for initializing the secrets stored at the server. Use the `!ServerPK` fact for clients to access the authentic server public key. Prove the lemma `Executability` that is already included as part of the source skeleton file.

Expected results

The lemmas `Executability` should be automatically provable. Tamarin will generate a warning should you forget to include one of the required action facts.

Part II: Security Properties

This challenge-response protocol intends to provide three security guarantees:

Secrecy The adversary cannot learn secrets stored at a server unless they compromise relevant key material. In particular, when a client key is compromised, secrets that have been fetched prior to the key compromise cannot be learned by the adversary. This guarantee is also called *forward secrecy* in the literature.

Agreement Whenever a client receives a secret, it was intended for them.

Injectivity of Agreement Secrets cannot be fetched twice, which provides replay protection.

Note that our model contains key compromise events, i.e., the rules `ServerCompromise` and `ClientCompromise` will reveal key material to the adversary, and your lemmas must account for that.

Task

Model all the above security properties in the three lemmas `Secrecy`, `Agreement`, and `Injectivity` while accounting for key compromise using the corresponding key compromise facts. The skeleton file already contains placeholder lemmas. Modify the associated trace property. Next to the action facts you added in the previous part, your properties can also reference the facts `StoreSecret`, `ServerRelease`, `ServerCompromise`, and `ClientCompromise`, which are already in the source skeleton file.
Model the protocol's forward secrecy guarantee in the `Secrecy` lemma as one of the cases accounting for key compromise. Do so by precisely stating when client key compromise can help the adversary learn a secret stored at a server.

Expected results

In its current form, the protocol suffers from a Meddler-In-The-Middle (MITM) attack. You should find that `Secrecy` is false, but `Agreement` and `Injective Agreement` to be true.

Part III: Fixing the Protocol

Finally, fix the protocol. The fix is simple. Only alter the exchanged messages. Do not add any new messages or new protocol steps.

When all properties are true, ensure that all cases in which you account for key compromise are *necessary*. By "necessary," we mean that you only forbid key compromise that is indeed required to violate the security property. For example, the easiest way to account for key compromise would be to assume `not Ex c #x. ClientCompromise(c) @ #x` in your properties. This would also make a property true when the adversary compromises client key material that is completely unrelated to your session, thus, unnecessary.

Task

Find and model a fix for the protocol such that all security properties are true. Then, ensure that you only forbid the necessary key compromises.

Expected results

With your fix implemented, **all** security properties should be true.

Submission

Complete the provided skeleton file following the tasks above and submit it at the following Moodle URL by **Monday, 13.10.2025, at 15:00**:

https://moodle-app2.let.ethz.ch/mod/assign/view.php?id=1262131.

You may submit as many times as you want. We will grade your **latest** submission and do not have access to earlier submissions. Thus, check carefully that you submitted the correct files and the correct versions. We use the same submission page for both parts of the lab.

The skeleton file is available at:

https://moodle-app2.let.ethz.ch/mod/resource/view.php?id=1280061.

Rules for Submission

- You may submit incomplete solutions, for example, the protocol without the fix.
- Develop the solution yourself, **without the help of others**.
- Name your solution file `fetching_protocol.spthy`.
- Do not submit folders or zip files.
- Use Tamarin version 1.10.0.
- **Do not use any restrictions**.
- Do not modify the executability lemma and rules that are already provided.
- Do not change the theory name, lemma or action fact names or the order of arguments for action facts.
- Do not use any action facts in the lemmas `Secrecy`, `Agreement`, and `Injectivity` that have not been introduced in this assignment.
- `tamarin-prover --prove fetching_protocol.spthy` must not raise any warnings or errors and must terminate.
- Test your submission using the script as explained below.

Testing your Submission

We provide a script that checks for some of the above requirements. Run this baseline test on each of your solutions before submission. **We will not accept solutions that fail this baseline test**, i.e. you will not receive any points for them. You can find the script at

`https://moodle-app2.let.ethz.ch/mod/resource/view.php?id=1280076`.

The script requires python3. You have to provide your path to Tamarin and the path to your submission to the script, i. e. by running:

```
./BaseLineScript <path-to-submission> $(which tamarin-prover)
```

If all tests pass, the script should output:

```
SUCCESS: You passed all the base line tests. You may submit your theory.
```

Otherwise you will get an error message. The script might ask for your path to your Maude installation as a third argument.

Evaluation

The grading will be fully automatic. For this second lab, you can receive 60 points. We test your lemmas against different models and we test your models against different lemmas. Each lemma receives points if it passes all our tests and each model receives points if it passes all our tests. We will not publish the test lemmas and models until after the submission deadline.

The point distribution is as follows:

	Model	Lemmas
`Protocol1.spthy`	7	3
`Protocol2.spthy`	7	3
`Protocol2Dict.spthy`	2	3
`Protocol3.spthy`	12	3
`fetching_protocol.spthy`	36	24