# Security Analysis of Cryptographic Protocols with DY*

**DY* Maintainer**

**Mar 20, 2025**

# CONTENTS


**1 Introduction** **3**

**2 Example Protocols** **5**
- 2.1 Two-message Protocol . . . 5
- 2.2 Online? . . . 6
- 2.3 Needham - Schroeder Protocol . . . 7
- 2.4 Attack on NS Protocol . . . 7
- 2.5 Needham - Schroeder - Lowe Protocol . . . 8
- 2.6 Login . . . 9

**3 Main Concepts in DY*** **13**
- 3.1 Communication Model . . . 13
- 3.2 Global vs Local View on Protocols . . . 14
- 3.3 States . . . 16
- 3.4 The Global Trace . . . 19
  - 3.4.1 Sending and Receiving of Messages . . . 21
- 3.5 Capabilities of Honest Participants . . . 21
- 3.6 Attacker Capabilities . . . 21
- 3.7 Stating Security Properties . . . 22
- 3.8 How To: Prepare Protocol Models for DY* . . . 22

**4 Modelling Protocols in DY*** **23**
- 4.1 Wire-format and Abstract Formats for Messages and Sessions . . . 23
  - 4.1.1 Abstract Formats for Messages . . . 23
  - 4.1.2 Abstract Formats for States . . . 24
  - 4.1.3 Translating between Abstract and Wire-Format . . . 26
- 4.2 The `traceful` and `option` Monads . . . 26
  - 4.2.1 The `traceful` Monad . . . 26
  - 4.2.2 The `option` Monad . . . 29
  - 4.2.3 The `traceful` + `option` Monad . . . 30
  - 4.2.4 Comparing `traceful` + `option` with `traceful` . . . 31
  - 4.2.5 Combining actions from different Monads . . . 32
- 4.3 A Model of the Two-Message Protocol . . . 35
  - 4.3.1 Setup . . . 35
  - 4.3.2 The abstract message and state types . . . 35
  - 4.3.3 The protocol steps . . . 40
  - 4.3.4 An Example Protocol Run . . . 49
- 4.4 How To: Modelling a Protocol in DY* . . . 52
- 4.5 A Model of the Online? Protocol . . . 56
  - 4.5.1 Public Key Encryption in DY* . . . 57

4.5.2 Setup . . . 60
4.5.3 The `Data` module . . . 60
4.5.4 The protocol steps . . . 63
4.5.5 An example protocol run . . . 71
4.6 A First Implementation of NSL . . . 81

**5 Stating Security Properties** **83**
5.1 Attacker Knowledge and Corruption . . . 83
5.1.1 Attacker Knowledge . . . 83
5.1.2 Corruption . . . 84
5.2 Secrecy . . . 84
5.3 Authentication . . . 88
5.4 How To: State Security Properties in DY* . . . 90

**6 Proving Security Properties** **93**
6.1 The Proof . . . 93
6.2 Labeling System . . . 93
6.3 Trace Invariants . . . 93
6.4 Proof of Secrecy for Online? Protocol . . . 93
6.5 Proof of Responder Authentication for Online? Protocol . . . 94
6.6 Proving Security Properties for NSL . . . 94
6.7 How To: Prove Security Properties in DY* . . . 94

DY* (pronounce as "D-Y Star") is a formal verification framework for the symbolic analysis of cryptographic protocols. The name consists of "DY" which stands for the underlying Dolev-Yao Model and the star (*) referring to the programming language it is written in: F*.

The framework accounts for advanced protocol features like unbounded loops and mutable recursive data structures, as well as low-level implementation details like protocol state machines and message formats, which are often at the root of real-world attacks.

DY* extends a long line of research on using dependent type systems for this task, but takes a fundamentally new approach by explicitly modeling the global trace-based semantics within the framework, hence bridging the gap between trace-based and type-based protocol analyses. This approach enables us to uniformly, precisely, and soundly model, for the first time using dependent types, long-lived mutable protocol state, equational theories, fine-grained dynamic corruption, and trace-based security properties like forward secrecy and post-compromise security.

DY* is built as a library of F* modules that includes a model of low-level protocol execution, a Dolev-Yao symbolic attacker, and generic security abstractions and lemmas, all verified using F*. The library exposes a high-level API that facilitates succinct security proofs for protocol code.

The code is available at GitHub.

In this document, we give a hands-on introduction to DY* as taught in a practical course for computer science students.

A document containing more details on the technical concepts and underlying theory of DY* can be found at TODO: link to detailed technical doc. The code examples used in this tutorial and all solutions can be found at TODO: link to GitHub Repo for this Tutorial.

CHAPTER

ONE

# INTRODUCTION

This is a hands-on introduction to DY* as taught in a practical course for computer science students.

We will use small example protocols to learn how to model a protocol in DY* and how to state and prove security properties. We assume knowledge of F* and refer to the F* Tutorial for an introduction.

In this chapter, we introduce the example protocols, and explain some of the main concepts of DY* on a high-level before going in to more details in the following chapters.

If you are already familiar with the Dolev-Yao symbolic model of protocols and trace based security properties, you may directly skip to *Modelling Protocols in DY** after the example protocols.

general intro on symbolic formal securtiy analysis of protocols

CHAPTER

TWO

# EXAMPLE PROTOCOLS

## 2.1 Two-message Protocol

In the two-message protocol (see Fig. *A simple Two Message Protocol*), Alice generates a nonce $n_A$ and sends this nonce together with her name to Bob. Bob then confirms that he received the message, by replying with the nonce $n_A$ he received.

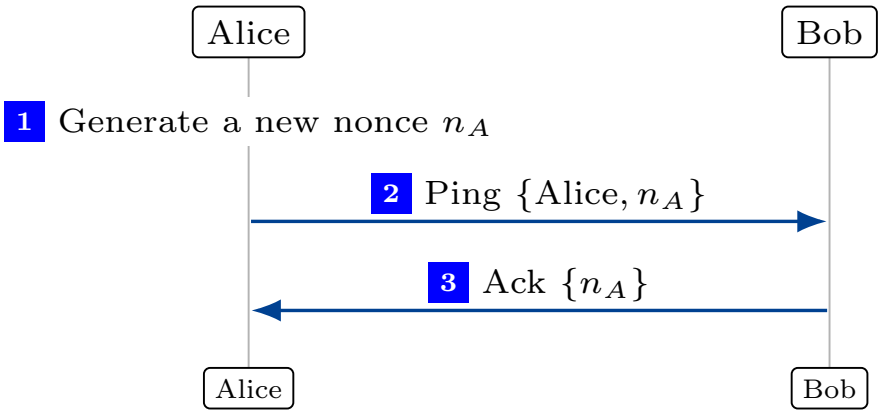


Fig. 1: A simple Two Message Protocol

This very simple protocol does not guarantee any security properties, we will use it just as illustrating example for the first part of the tutorial.

The two-message protocol can be made a bit more secure by encrypting the messages for the corresponding recipients. This is then called the Online? protocol.

## 2.2 Online?

With the Online? protocol (see Fig. *The Online? Protocol*) Alice wants to figure out whether Bob is online. She creates a nonce $n_A$ and sends it together with her name to Bob. This message is encrypted with Bob's public key. If Bob is online and receives this message, he'll reply by sending back the nonce $n_A$ encrypted for Alice.

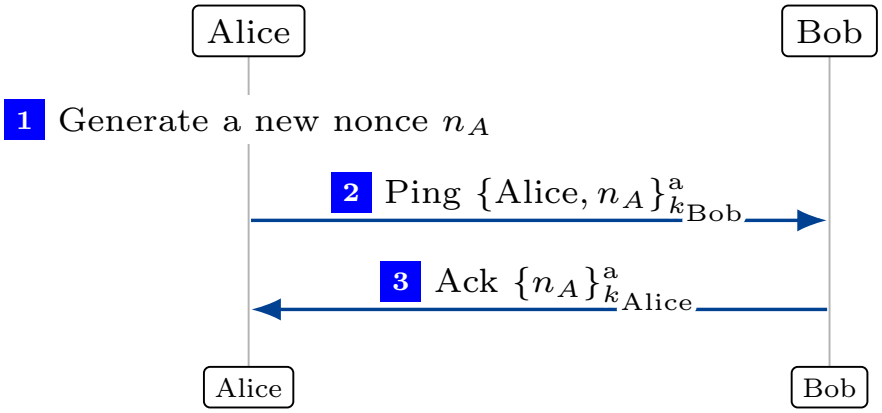


Fig. 2: The Online? Protocol

Using encryption gives us a one-sided *secrecy property*: A nonce $n_A$ that Alice generates for (and sends to) some honest other party Bob, is only known to Alice and Bob.

However, we don't have secrecy of a nonce that Bob receives, i.e., it is not true that any nonce Bob receives from Alice, is only known to Alice and Bob. An attacker can do a MitM attack, sitting between Alice and Bob (see Fig. *A MitM attack on the Online? Protocol breaking Bob-sided secrecy*):

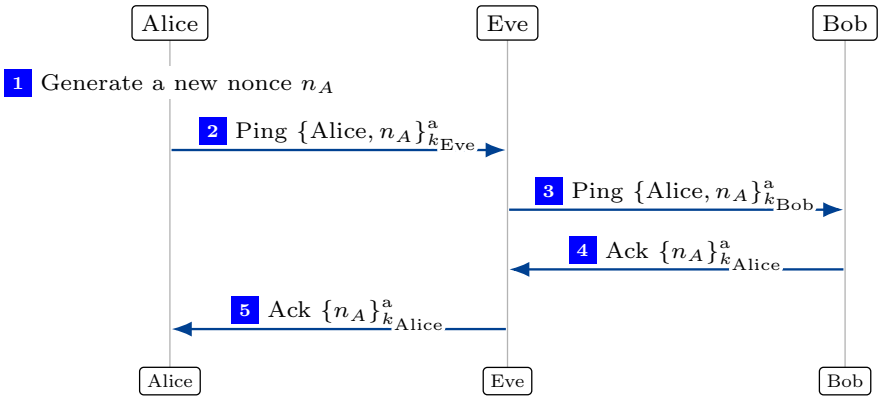


Fig. 3: A MitM attack on the Online? Protocol breaking Bob-sided secrecy

In this attack, Bob thinks that he is talking to Alice, and hence that the nonce is only known to Alice and Bob. However, the nonce is also known to the attacker Eve.

Note: The secrecy property from Alice's perspective is satisfied! The property only talks about *honest* parties. Intuitvely speaking: If you send something willingly to the attacker (as Alice does in this case), it is no longer a secret.

As a second property, we consider *responder authentication*: If Alice at the end of a run believes, she talks with Bob, then this Bob must have been involved in the run, i.e., Bob must have sent a response.

Note, that we don't have *initiator authentication*: In the MitM attack, Bob believes to be talking with Alice, but instead he is talkin with Eve and Alice is not involved in the run with Bob.

## 2.3 Needham - Schroeder Protocol

As a second example we consider the Needham-Schroeder (NS) protocol (see Fig. *The NS protocol*) used to agree on a shared secret between two participants.

First, Alice creates a new nonce $n_A$ and sends it encrypted for Bob. Now Bob also creates a new nonce $n_B$ and replies to Alice's message with both nonces $n_A$ and $n_B$, encrypted for Alice. Finally, Alice sends back $n_B$ encrypted to Bob. The nonce $n_B$ can now be used as a shared secret between Alice and Bob.

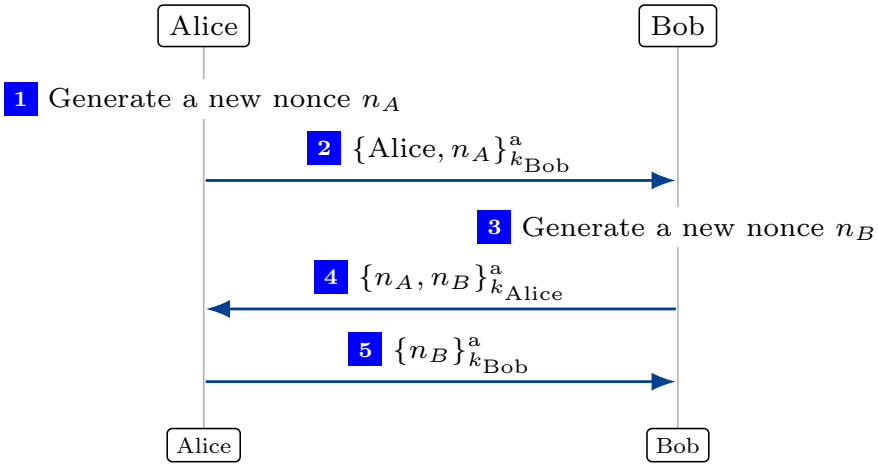


Fig. 4: The NS protocol

We would like to have a secrecy property stating that $n_B$ is only known to Alice and Bob after a successful run of the NS protocol. Unfortunately, there is a man-in-the-middle attack destroying the secrecy of $n_B$ (see Fig. *Attack on the NS Protocol*).

## 2.4 Attack on NS Protocol

In this flow, Bob thinks he is talking with Alice and he doesn't notice anything wrong from his point of view, but the nonce $n_B$ he generates (intended for Alice) leaks to Eve.

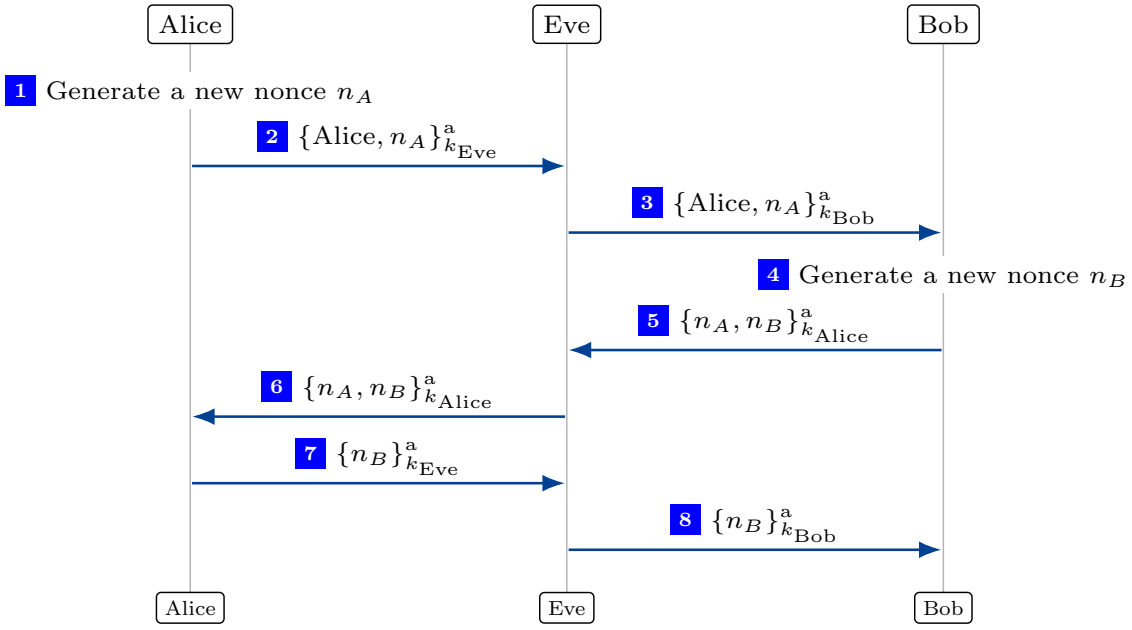


Fig. 5: Attack on the NS Protocol

## 2.5 Needham - Schroeder - Lowe Protocol

The previous attack can be prevented by a small change to the protocol. Instead of just sending the two nonces, Bob also sends his name. This is then the Needham-Schroeder-Lowe (NSL) protocol (see Fig. *The NSL protocol*):

Alice generates a nonce $n_A$ and sends it together with her name encrypted to Bob. Bob then generates his own nonce $n_B$ and sends the two nonce $n_A$ and $n_B$ together with his name to Alice. This message is encrypted for Alice. Finally, Alice sends the nonce $n_B$ back to Bob.

The MitM attack is prevented, since Eve can not just forward Message 4 to Alice. Alice expects to see Eve's name in there and not Bob. So Alice would stop the protocol run at that point. Note that Eve can not change the name in the message, since Bob sends this message encrypted *for Alice*.

Indeed, this protocol guarantees secrecy of both nonces $n_A$ and $n_B$, i.e., the attacker does not get to know the nonces as long as both Alice and Bob are honest.

Additionally, the NSL protocol also provides *responder authentication*, similar to the Online? protocol: If Alice at the end of a run believes to be talking with Bob, then this Bob must indeed be involved in the run.

The Online? protocol did not provide the reverse direction of *initiator authentication*. The NSL protocol does: If Bob at the end of a run believes to be talking with Alice, then this Alice must indeed be involved in the run.

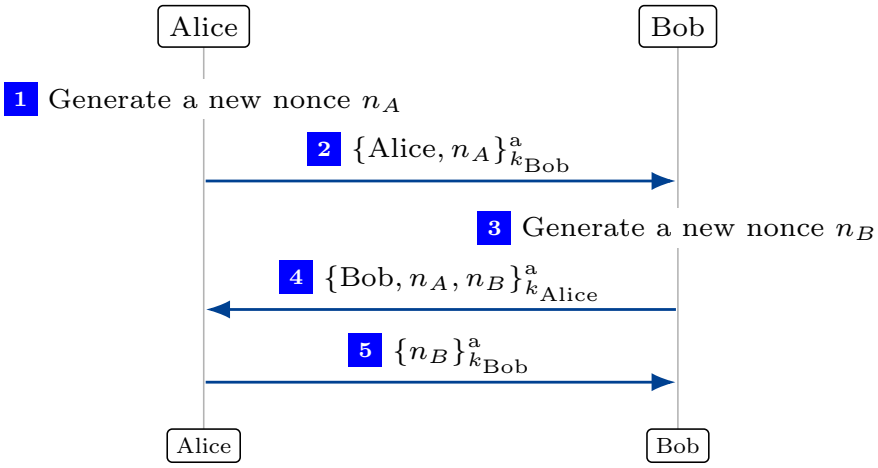


Fig. 6: The NSL protocol

## 2.6 Login

Our final example is a simple registration and login protocol (just called Login protocol) shown in Fig. *The Login Protocol*. It has two phases: a registration and a login phase.

For registration, the client first chooses and stores a new password. It then generates a nonce $N_C$ and sends a message containing the tag "42", the nonce $N_C$ and the password together with the name of the client. This message is sent asymmetrically encrypted to the server.

The server now checks whether it already has a client with name $C$ and the password in its database. If not, it creates a new account by storing the name and password, and a newly generated cookie. It then replies with a message containing the tag "1337" and the string "ok". This message is symmetrically encrypted with the nonce $N_C$. The client now knows that it has an account at the server and can log in.

For logging in, the client generates a new nonce $N'_C$ and sends a message containing the tag "42", the nonce $N'_C$ and the password together with its name. If the server already has a client with the password in its database, it reads the corresponding cookie from the account and replies with a message containing the tag "23" and the cookie. This message is symmetrically encrypted with the nonce $N'_C$. The client stores the cookie and the protocol ends.

Note that the messages sent by the client (Message 2 and Message 6) are the same for registration and log in. For registration the client chooses a new password and for log in it chooses some existing password. The server expects only one type of message and can't immediately see whether this is a registration or a log in message. To react correctly, it has to check the stored accounts.

For this protocol we want to show a secrecy property for the cookie from the server's point of view: A cookie stored at the server is only known to the server and the client it was created for.

**Goals of the Course**

- Model and Implement the Two-Message protocol
- Model and Implement the Online? protocol and show Alice-sided secrecy
- Model and Implement the NS Protocol and the Attack of Lowe
- Model and Implement the NSL Protocol and show the secrecy property

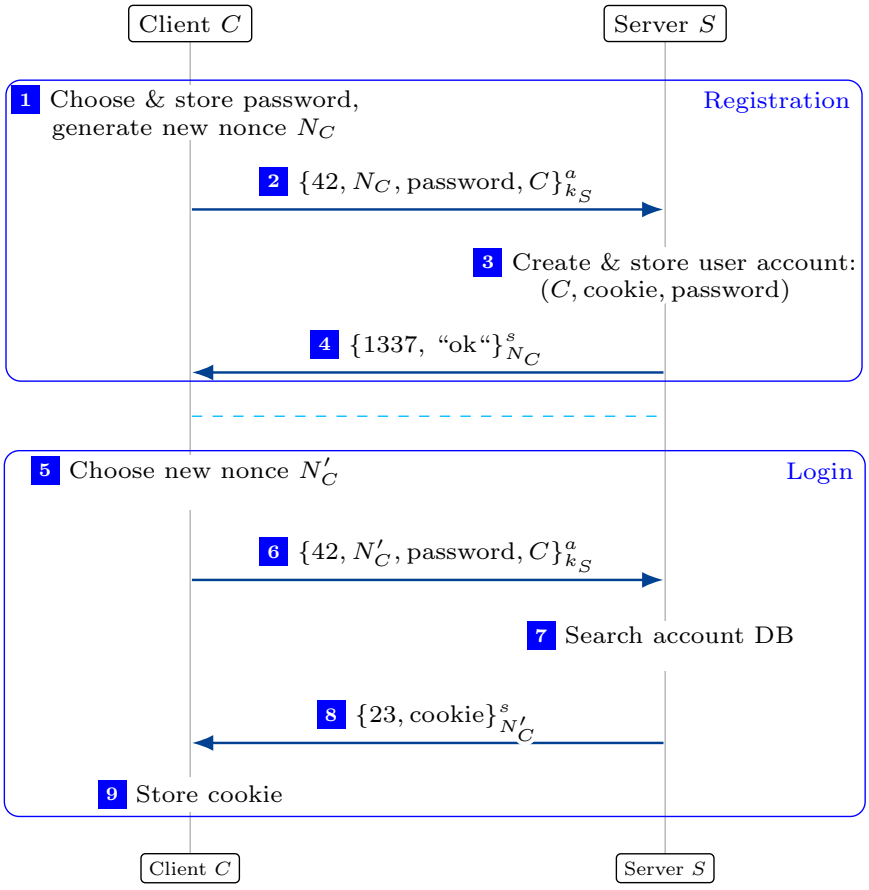


Fig. 7: The Login Protocol

- Model and Implement the Login Protocol and show the secrecy property

CHAPTER

# THREE

# MAIN CONCEPTS IN DY*

## 3.1 Communication Model

In the Dolev-Yao model, sending and receiving of messages is asynchronous. A message from $A$ to $B$ is not sent directly from $A$ to $B$, instead $A$ hands the message to the (delivery) network, who takes care of delivering the message to $B$ at some later point. Think of this in the same way a letter is delivered: If Alice sends a letter to Bob, she puts the letter in a post box, handing it to the postal service who delivers the letter to Bob's letter box.

In this asynchronous model, the delivery network plays a central role in scheduling the delivery of messages and delivering messages to their intended receivers.

If we look at the descriptions of our example protocols in *Example Protocols*, they are given as a sequence of messages sent from one participant to another. But since we use asynchronous communication in DY* we need to reformulate these descriptions to match the communication model.

We split each communication via a message into two steps: a send step and a *receive-react* step.

**Example: Asynchronous Description of Two-Message Protocol**

A description of the Two-message protocol (see *Two-message Protocol*) in the asynchronous communication model is as follows:

1. *Initiate flow and send first message:*

   Alice generates a nonce $n_A$ and sends it together with her name to Bob

2. *Receive first message and send second message:*

   Bob receives the nonce $n_A$ from Alice and sends it back to Alice

3. *Receive last message:*

   Alice receives $n_A$ from Bob and finishes the protocol flow

**Example: Asynchronous Description of Online? Protocol**

A description of the Online? protocol (see *Online?*) in the asynchronous communication model:

1. *Initiate flow and send first message:*

   Alice generates a nonce $n_A$, encrypts the nonce together with her name for Bob, and sends the message to Bob

2. *Receive first message and send second message:*

Bob receives the nonce $n_A$ from Alice, encrypts it for Alice and sends it back to Alice

3. *Receive last message:*

   Alice receives $n_A$ from Bob and finishes the protocol flow

**Exercise: Asynchronous Description of Needham-Schroeder Protocol**

Adapt the description of the Needham-Schroeder protocol from *Needham - Schroeder Protocol* to match the asynchronous communication model.

**Show/Hide Answer**

1. *Initiate flow and send first message:*

   Alice generates a nonce $n_A$ and sends it (together with her name) encrypted for Bob

2. *Receive first message and send second message:*

   Bob receives the nonce $n_A$ from Alice, generates a new nonce $n_B$ and sends both nonces encrypted for Alice

3. *Receive second message and send third message:*

   Alice receives the two nonces $n_A$ and $n_B$ from Bob and sends back $n_B$ encrypted for Bob

4. *Receive last message:*

   Bob receives $n_B$ and finishes the protocol flow

## 3.2 Global vs Local View on Protocols

A second point where the intuitive description of a protocol differs from the model in , is that the *order* of the protocol steps is *not* part of the model. Whereas in *Example Protocols* we describe *one* flow of a protocol with a fixed sequence of messages, in we want to model arbitrarily *many*, possibly *interleaving* flows running in parallel. For this we model each receive-react step from the previous subsection separately, allowing steps to be executed at any time (possibly in different flows).

Following this approach we need to think of how to identify flows and keep track of their state. If we just had the receive-react steps of the Needham-Schroeder protocol as in the previous subsection, we could, for example, send several messages to Alice including two nonces, and Alice would reply to all of those. However, this does not correctly model the NS protocol. Alice should only react to such a message, if she previously initiated a flow with one of the nonces. We resolve this issue by tracking the state of protocol flows *locally* at every participant. In our example, Alice would keep track of flows she initiated and respond to a message with two nonces, only if she has a corresponding ongoing flow.

This is in contrast to the intuitive description of protocols as in *Example Protocols*, where the state of flows is tracked *globally* by the specified sequence of messages.

**Example: Local View of Two-Message Protocol**

We adapt our description of the Two-message protocol from *Two-message Protocol* to match the local view on the protocol used in DY*.

To keep track of the state of protocol flows, we store successful completion of each step. We identify a flow

- at the initiator by the nonce contained in the first message and the peer it is sent to, and

- at the responder by the nonce received in the initial message and the peer it received the message from.

With this we adapt the receive-react steps from the previous example as follows:

- *Initiate flow and send first message:*

  Alice generates a nonce $n_A$ and sends it together with her name to Bob.

  She stores that she initiated a flow with Bob using $n_A$.

- *Receive first message and send second message:*

  Bob receives a message of the form $(n_A, \text{Alice})$. He sends the nonce $n_A$ back to Alice.

  Bob stores that he received $n_A$ in a first message from Alice and replied.

- *Receive last message:*

  Alice receives a nonce $n_A$.

  If she previously initiated a flow using $n_A$, she stores that she received a response and finishes this protocol flow.

**Example: Local View of Online? Protocol**

We get the local view description of the Online? protocol from *Online?* in the same way as for the two-message protocol.

**Exercise: Local View of Needham-Schroeder Protocol**

Adapt the description of the Needham-Schroeder protocol in *Needham - Schroeder Protocol* to match the local view on the protocol used in .

**Show/Hide Answer**

To keep track of the state of protocol flows, we store successful completion of each step. We identify a flow at the initiator by the nonce contained in the first message and the peer it is sent to, and at the responder by the two nonces sent in response to the initial message.

- *Initiate flow and send first message:*

  Alice generates a nonce $n_A$ and sends it encrypted for Bob.

  She stores that she initiated a flow with Bob using $n_A$.

- *Receive first message and send second message:*

  Bob receives a message of the form $(n_A, \text{Alice})$. He generates a new nonce $n_B$ and sends both nonces encrypted for Alice.

  Bob stores that he received a first message from Alice and replied with $n_A$ and $n_B$.

- *Receive second message and send third message:*

  Alice receives two nonces $n_A$ and $n_B$.

  If she previously initiated a flow using $n_A$ with a corresponding Bob, she sends back $n_B$ encrypted for Bob and stores that she received two nonces and replied.

- *Receive last message:*

  Bob receives a nonce $n_B$.

If he previously received a first message and replied with $n_B$, he finishes the protocol flow.

A key task when modeling protocols in the local view, is to figure out what information is needed at each participant to continue with subsequent steps.

For example, consider the "Initiate Flow" step above. The first idea might have been to just store the nonce $n_A$ at Alice. But, when she later receives the response, she needs to send the third message with $n_B$ to someone. Recall that in the asynchronous communication model, she does not get information about the sender of the second message. So she can not directly reply. Thus, she needs to store who she sent the initial message to, to now send the third message to the same participant.

**Remember**

A protocol model in DY*

- … is split into several receive-react steps.
- … does not contain information about the *order* of these steps.

So don't think of the *global* protocol flow as given in an intuitive description, but rather think of each receive-react step *locally* at each participant.

We will now take a closer look at the main concepts used in DY* for tracking the state of protocol flows and sending and receiving of messages.

## 3.3 States

To store information locally, every participant has a *state*. This state consists of several *sessions* that contain some information.

In the simplest case, one session in a state corresponds to one run of a protocol.

**Example: States in *one* run of the Two-Message Protocol**

We look at the states of Alice and Bob in one run of the Two-Message protocol.

When Alice initiated a run her state looks like this:

```
State Alice:
  Session 0: SentPing n_A to Bob
```

There is one session where she stores the status of the protocol run. Here she stores that she initiated a run with Bob sending the nonce $n_A$.

When Bob received such a ping and replied, his state looks like this:

```
State Bob:
  Session 0: SentAck n_A to Alice
```

He also has one session for keeping track of the protocol status.

Finally, when Alice received the Ack of Bob, her state changes to:

```
State Alice:
  Session 0: ReceivedAck n_A from Bob
```

In this case, Alice considers Session 0 to be finished.

To illustrate the concept of one session per run at each participant even more, we look at a slightly more complex example, where we consider several *interleaving* runs of the Two-Message protocol.

**Example: States in *interleaving* runs of the Two-Message Protocol**

Alice, Bob and Charlie are running several runs of the Two-Message Protocol in parallel. These runs are stored in different sessions in their states.

At some point, their states could look like this:

```
State Alice:
  Session 0: ReceivedAck n_A1 from Bob
  Session 1: SentPing n_A2 to Bob
  Session 2: SentPing n_A3 to Charlie

State Bob:
  Session 0: SentPing n_B1 to Charlie
  Session 1: SentAck n_A1 to Alice

State Charlie:
  Session 0: SentAck n_A3 to Alice
  Session 1: SentAck n_B1 to Bob
```

At this point, we have the following situation:

- Alice has 3 ongoing runs of the protocol:
  - in Session 0 she finished a run with Bob
  - in Sessions 1 and 2, she initiated two more runs with Bob and Charlie resp.
- Bob has 2 ongoing runs:
  - in Session 0 he initiated a run with Charlie
  - in Session 1 he replied to Alice (this corresponds to Alice's Session 0 since they are both identified by the nonce $n_{A1}$)
- Charlie has also 2 ongoing runs:
  - in Session 0 he replied to Alice (which in turn correspond to Alice's Session 2 ($n_{A3}$))
  - in Session 1 he replied to Bob (corresponding to Bob's Session 0 ($n_{B1}$))

In the simple Two-Message protocol, we only have sessions keeping track of protocol runs. However, sessions can also be used to store global information at a participant that is independent of the individual protocol runs. Such global information are for example private keys for decryption and public encryption keys. We see this in the Online? protocol:

**Example: Sessions with global Information in the Online? Protocol**

After the first step of one run of the Online? protocol, Alice's state looks like this:

```
State Alice:
  Session 0: private decryption key k_A
  Session 1: public encryption key pk_B of Bob
  Session 2: SentPing n_A to Bob
```

We see, that Session 2 is the session keeping track of the protocol run. Just as in the previous examples for the Two-Message protocol.

New are the first two sessions that now store some global information: Session 0 contains Alice's private decryption key $k_A$ and Session 1 a public encryption key $k_B$ for Bob.

The state of Bob at the same time (before receiving the first message from Alice) is:

```
State Bob:
  Session 0: private decryption key k_B
  Session 1: public encryption key pk_A of Alice
```

He just stores his private key and the public key of Alice.

To keep track of the current state of a protocol run, participants need to be able to *update* existing sessions. Consider the previous example: When Alice receives the Ack from Bob, she needs to update her Session 2 to store this progress in the run.

On the other hand, participants must also be able to *create* new sessions. Again, in the previous example, we see that Bob does not yet have any session for the started run. Once he receives the Ping from Alice and replies with an Ack, he needs to store the progress of the run. So he must be able to create a new session.

And indeed, these are the two operations on session that are available:

- add a new session to a state
- update an existing session

Comparing the different sessions for protocol runs and global information in terms of how often they are upated, we observe, that we update protocol sessions frequently, usually after every protocol step of the participant. In contrast, the sessions containing global information are mostly set up at the very beginning and don't change after that.

This intuition of one session per protocol run that is updated after every protocol step and some sessions for global information that are only set up in the beginning is sufficient for now.

**Remember**

A session in a state of a principal can store

- global information like private or public keys
- the current state of a protocol run

New sessions can be added to a state and existing sessions can be updated.

You should now be able to explain how the states of Alice and Bob look like in a run of the Needham-Schroeder protocol:

**Exercise: States in the Needham-Schroeder Protocol**

How do the states of Alice and Bob look like in a run of the Needham-Shroeder protocol?

Show them at the beginning of the run, and after each protocol step. Look back at *Exercise: Local View of Needham-Shroeder* to recall the information that needs to be stored in the states.

**Show/Hide Answer**

The initial states of Alice and Bob before the run begins, just contain the corresponding private and public keys:

```
State Alice:
  Session 0: private decryption key k_A
  Session 1: public encryption key pk_B of Bob

State Bob:
  Session 0: private decryption key k_B
  Session 1: public encryption key pk_A of Alice
```

After Alice started the run, she adds a new session for the state of this run:

```
State Alice:
  ...
  Session 2: SentMsg1 n_A to Bob
```

When Bob replied to this 1st message, he also creates a new session:

```
State Bob:
  ...
  Session 2: SentMsg2 (n_A, n_B) to Alice
```

When Alice receives this response and in turn responded with the last message, she updates Session 2:

```
State Alice:
  ...
  Session 2: The run for (n_A, n_B) with Bob has finished
    and I sent Msg3 n_b to Bob
```

And finally, after Bob received this message, he also updates his session for this run:

```
State Bob:
  ...
  Session 2: The run for (n_A, n_B) with Alice has finished
    as I received Msg3 (n_B) from Alice
```

## 3.4 The Global Trace

The overall state of the system, including all messages sent so far and the states of participants across all parallel flows, are captured on **the trace**, which is a log of observable protocol actions. The trace is an ordered collection of **entries** of the following types:

- **messages** sent between participants
- **states** of participants
- **events** logged by participants (for example, "Alice initiated a flow with Bob")
- generation of **random values** (for example, keys and nonces)
- **corruption** (more on the attacker model and corruption later)

Since the trace acts as a log, entries can only be *appended* to the end of the trace, but not removed from the trace or changed later on.

**Example: Trace of Run of Two-Message Protocol**

The trace after a successful run of the Two-Message protocol looks like this:

```
1. Generate nonce n_A
2. Message: Ping (Alice, n_A)
3. Session 0 of Alice: SentPing n_A to Bob
4. Message: Ack n_A
5. Session 0 of Bob: SentAck n_A to Alice
6. Session 0 of Alice: ReceivedAck n_A from Bob
```

**Example: Key Setup for Online? Protocol**

The key setup phase for the Online? Protocol is captured on the trace like this:

```
1. Generate private key k_A for Alice
// Alice stores this private key
2. Session 0 of Alice: Private Key k_A

3. Generate private key k_B for Bob
// Bob stores his private key
4. Session 0 of Bob: Private Key k_B

// Alice stores the public key of Bob
5. Session 1 of Alice: Public Key pk_B of Bob
// Bob stores the public key of Alice
6. Session 1 of Bob: Public Key pk_A of Alice
```

Observe from the previous example, that a trace entry only contains *one* action and not the complete global state where relevant parts are updated. In particular for states that means that a trace entry contains only one state (TODO: state vs session vs version: What terms to use?), and not the full state of a principal. For example, in Step 5, Alice adds a new Session 1 to her state. The corresponding trace entry captures only the new state, and not the *full* state of Alice.

**Example: Trace for Attack on Needham-Schroeder Protocol**

The beginning of the trace corresponding to the attack flow for the Needham-Schroeder protocol looks like this (simplified):

```
// Setup Phase: Key generation and compromising Eve
0.  Generate a private decryption key for Alice
1.  Store the private key in Alice's state
2.  + 3. Generate and store a private decryption key for Bob
4.  + 5. Generate and store a private decryption key for Eve
6.  Compromise the state session where Eve stores her private key
7.  Store the public encryption key of Alice in Bob's state
8.  Store the public encryption key of Bob in Eve's state
9.  Store the public encryption key of Eve in Alice's state
10. Compromise the state session where Eve stores Alice's encryption key

// Actual Flow - Step 1: Alice starts a flow with Eve and sends a nonce $n_a$␣
↪encrypted to Eve
11. Generate nonce n_a
12. Event: Alice initiated flow with Eve using n_a
13. Store (Eve, n_a) in Alice's state
14. Generate a nonce n_pke used for encryption
```

```
15. Message from Alice to Eve: encrypted (n_a, Alice) with Eve's key and used n_pke
...
```

In the above trace part we see trace entries of all 5 types:

- messages: entry 15,
- states of participants: entries 1,3,5,7,8,9,13,
- an event: entry 12,
- generation of random values: entries 0,2,4,11,14, and
- corruption: entries 6 and 10.

### 3.4.1 Sending and Receiving of Messages

Among other things, the trace is used as post and letter box for our asynchronous communication model. To send a message, a new message entry is added to the trace and receiving a message is just reading the corresponding message entry. Delivery of the message by the postal service corresponds to telling the receiver which message entry to read.

## 3.5 Capabilities of Honest Participants

Honest participants can read and write sessions of their own state, send and receive messages, create events, and generate nonces.

## 3.6 Attacker Capabilities

Recall that we consider a Dolev-Yao network attacker, who controls the network (handling of messages) and can corrupt participants. In we model this by specifying how the attacker can access the trace.

**Messages**
: The attacker can read all message entries on the trace. He also acts as the postal service delivering messages. Recall, that receiving a message is reading a specific message entry from the trace. The attacker chooses which message entry is to be read by whom at what point (if at all). He can also create and send messages on his own.

**States**
: The attacker can corrupt single sessions of a participant state by adding a corruption event to the trace. He can then read the content of all corrupted sessions.

From this information that the attacker can directly read off the trace, he can also derive new information and use this to create own messages or state sessions. For example, if he knows two values $x$ and $y$ from some messages on the trace, he can produce a new message containing both values. We give full details on deriving new values later, for now the intuition that the attacker can do anything an honest participant can do with information he knows, should suffice. Most notably, no one (not even the attacker) can break cryptography. For example, if the attacker sees a message that is encrypted with some key that the attacker doesn't know, he can't access the plain text of the message.

## 3.7 Stating Security Properties

Our security properties are trace-based properties and talk about all traces that follow the protocol specification. For example, secrecy properties are often of the form "In any trace following the protocol specification, a secret value between Alice and Bob is not known to the attacker, if neither Alice nor Bob are corrupted."

**Example: Secrecy Property for NSL**

For the NSL protocol we want to show secrecy of the shared secret key $n_B$. The property is stated in in the following simplified form:

For all traces following the protocol specification and all values $n_B$ the following holds: if Bob finishes a flow with Alice and $n_B$, then the attacker does not know $n_B$ or one of Alice and Bob is corrupted.

In *[chap:labelingsystem,chap:proofmethod]*, we give more details on security properties in DY* and in particular, how to state that a trace follows the protocol specification.

## 3.8 How To: Prepare Protocol Models for DY*

**How-To: Prepare Protocol Model for DY***

1. Reformulate the intuitive description of the protocol to a local description using *receive-react steps*
2. For each of the receive-reat steps think about:
   - Who is the active participant?
   - What are the *actions* performed
     - What messages are sent?
     - Is there a change in state?
   - What *information* has to be stored in the states of participants for them to continue the next steps?

CHAPTER

# FOUR

# MODELLING PROTOCOLS IN DY*

In this section, we will start to implement the models of our simple example protocols in DY*. But first, we need to look at two more concepts,

## 4.1 Wire-format and Abstract Formats for Messages and Sessions

TODO: should this concept move to the concepts section? (without code records)

On the lowest level of a real network, messages are sent in a wire-format which can be thought of as a byte-stream without any structure. But the different network layers interpret this stream and give it some structure to make processing on higher layers easier. For example a UDP message contains some header that has several fields with meta-information followed by the actual content of the message. Further, the higher layers don't really care about or even need to know how their message is encoded on the lower layers. They just work with the content they need.

In DY* we use the same idea: The content of entries on the trace are stored in the wire-format, which is called `bytes`. But if we implement a protocol we don't want to work with the wire-format, instead we are only interested in the *content* of the messages. We collect the different components of our messages in an *abstract format*.

### 4.1.1 Abstract Formats for Messages

**Example: Abstract Message Format for the Two-Message Protocol**

The first message of the Two-Message Protocol (the Ping) has the following abstract structure: it consists of two entries, one of which is the name of Alice and the other one is some nonce.

The second message (the Ack) contains only one entry: the nonce.

In our implementations, we use F*'s record type for abstract message formats.

**Example: The records for the Abstract Format for the Two-Message Protocol**

We define a new type for the Ping message, collecting the two entries:

```
type ping_t = {
  alice: principal;
  n_a : bytes;
}
```

The name of Alice is captured in the `alice` component. It is a field of type `principal` (which is just a `string`).

The nonce is caputured in the `n_a` component. This field is of type `bytes`.

Similarly, we define a new type for the Ack message:

```
type ack_t = {
  n_a : bytes;
}
```

**Example: Abstract Message Format for the Online? Example**

The messages of the Online? protocol, have the same structures as the messages for the Two-Message protocol. We can thus use the same record types.

**Exercise: Abstract Message Format for the Needham-Schroeder Protocol**

Describe the abstract message format for the Needham-Schroeder protocol and define the corresponding record types.

**Show/Hide Answer**

The first message consists of two parts: the name of Alice, and a nonce. Similarly to before, we define the record:

```
type message1_t = {
  alice: principal;
  n_a : bytes;
}
```

The second message also has two parts: the two nonces. We define the record:

```
type message2_t = {
  n_a : bytes;
  n_b : bytes;
}
```

The final message contains only the nonce of Bob:

```
type message3_t = {
  n_b : bytes;
}
```

### 4.1.2 Abstract Formats for States

We do the same for states: States are also stored on the trace in the wire-format, however, we want to work with a more structured abstract format.

**Example: Abstract State Format for the Two-Message Protocol**

In the Two-Message protocol, we have three states (recall from *Example: States in one run of the Two-Message Protocol*):

- The first state (SentPing) consists of two parts: the nonce $n_A$ and the name of Bob.

- The second state (SentAck) also consists of two parts: the nonce $n_A$ and the name of Alice.
- Finally, the third state (ReceivedAck) also consists of two parts: the nonce $n_A$ and the name of Bob.

This describes the abstract state format.

**Example: The records for the Abstract State Format for the Two-Message Protocol**

From the abstract state format, we define the record types:

```
type sent_ping_t = {
  bob : principal;
  n_a : bytes;
}
```

```
type sent_ack_t = {
  alice: principal;
  n_a : bytes;
}
```

```
type received_ack_t = {
  bob : principal;
  n_a : bytes;
}
```

**Exercise: Abstract State Format for the Needham-Schroeder Protocol**

Describe the abstract state format for the Needham-Schroeder protocol and define the corresponding record types. (Refer to *Exercise:States in the Needham-Schroeder Protocol* for the states.) Consider *only* the states storing the progress of the protocol. (I.e., ignore the states for storing keys.)

**Show/Hide Answer**

There are 4 kinds of states for storing the protocol progress:

- The state, after Alice sent the first messge (SentMsg1): it contains two parts: the nonce $n_A$ and the name of Bob
- The state after Bob replied with a second message (SentMsg2): it contains three parts: the two nonces $n_A$ and $n_B$ and the name of Alice
- The state after Alice receives the third message from Bob: it contains three parts: the two nonces and the name of Bob
- The state after Bob received the third message from Alice: it stores the same things as the previous state of Bob: the two nonces and the name of Alice

We define the record types:

```
type sent_msg1_t = {
  bob : principal;
  n_a : bytes;
}
```

```
type sent_msg2_t = {
  alice : principal;
  n_a : bytes;
  n_b : bytes;
}
```

```
type sent_msg3_t = {
  bob : principal;
  n_a : bytes;
  n_b : bytes;
}
```

```
type received_msg3_t = {
  alice : principal;
  n_a : bytes;
  n_b : bytes;
}
```

### 4.1.3 Translating between Abstract and Wire-Format

Once we have our abstract message and state formats, we need to take care of transforming this abstract format to the wire-format and vice versa. For example, when sending a message and when receiving a message, we need to translate from the abstract message format to the wire-format and vice versa, respectively. Similarly, when we want to store a state on the trace, we need to translate from the abstract state format to the wire-format. We call the translations *serializing* (abstract to wire-format) and *parsing* (wire to abstract format).

We use a tool called Comparse for translating between abstract and wire-format. We will not go into the details of how Comparse works. We just use it and explain everything necessary for that.

**Remember**

- Everything stored on the trace, is stored using the *wire-format* `bytes`.
- In a protocol model, we work with *abstract formats* for messages and states, i.e., just the contents.
- Translating from abstract to wire-format is called *serializing*
- Translating from wire to abstract format is called *parsing*

## 4.2 The `traceful` and `option` Monads

### 4.2.1 The `traceful` Monad

As we have seen previously, the core object for capturing the overall state and for stating and reasoning about properties of a protocol, is the *global trace*.

Every protocol step is a sequence of reading entries from the current trace, or adding new entries to the trace. Each action in this sequence works with the new trace resulting from the previous action.

**Example: Explicit Trace Manipulation in the Two-Message Protocol**

The explicit trace manipulations of the first step of the Two-Message protocol (see *Example: Local View of Two-Message Protocol*) is:

0. The current trace before the step begins is `tr_in`
1. Generate a nonce $n_A$ and store it on the current trace `tr_in`. This results in the new trace `tr_nonce`.
2. Send the first message, i.e., append the message to the current trace `tr_nonce`. This results in the new trace `tr_message`.
3. Store the SentPing state, i.e., append the new state to the current trace `tr_message`. This results in the new trace `tr_state`.

The whole first step transforms the input trace `tr_in` to the final trace `tr_state`.

This explicit trace passing from one action to the next, is rather cumbersome and we want to avoid doing that manually. DY* offers the `traceful` monad for *implicit* trace handling. With this, we can just write the sequence of actions *without* having to think of the underlying traces.

At this point, you do *not* need to know what a monad is. We will see everything we need to *use* the monad in the following. The important general intuition is that monadic actions compute some return value (just like normal functions) but may have *side effects*. In our case that means, a traceful action works with the trace and may change the trace. For example, sending a message has the side effect, that the message is added to the trace.

**Remember: Intuition for Monads**

Monadic actions are functions with *side effects*.

Specifically, a *traceful* action may look at and change the current trace.

**Info on Monads**

If you want to understand the general concept of monads in F*, we refer to the corresponding chapter in the F* Book.

**If you are familiar with monads**

The `traceful` monad is a *state monad*, where the state is the trace with the additional condition, that the new trace can only append entries to the old trace. The value/result type can be anything.

In the previous example, we see that each step has an effect on the trace (i.e., changes the trace). We can thus consider the steps as traceful actions.

**Example: Implicit Trace Manipulation in the Two-Message Protocol**

Using the `traceful` monad, we can write the first step of the Two Message protocol as:

1. Generate a new nonce $n_A$ (this implicitly changes the trace)
2. Send the first message (this implicitly takes the trace after action 1 and produces a new trace)
3. Store the SentPing state (this implicitly takes the trace after action 2 and produces a new trace)

Which is exactly our intuitive description from *Example: Local View of Two-Message Protocol*.

### Sequential composition of traceful functions

Combining two `traceful` actions sequentially, like steps 1., 2. and 3. in the previous example, is done with `;*`: `traceful_action_1 ;* traceful_action_2`.

**If you are familiar with monads**

`a;* b` is syntactic sugar for `bind a (fun _ -> b)`

**Example: Sequential Composition of Traceful Functions**

The following code snippet

```
send_msg message ;*
store_state alice state
```

executes two traceful actions sequentially:

1. A message (`message`) is sent (i.e., added to the end of the current trace)
2. A new state (`state`) is stored for `alice` (i.e., the state is added to the trace after the message)

### Accessing return values of traceful functions

To access the computed value of a traceful action, we use `let*`:

```
let* return_value = traceful_action in
// now we can use return_value in subsequent actions
```

**If you are familiar with monads**

`let* x = a in b` is syntactic sugar for `bind a (fun x -> b)`

**Example: Accessing Return Values of Traceful Functions**

With `let* n_a = gen_rand in` we can access the nonce, returned by the traceful `gen_rand` function and use it later on under the name `n_a`.

*Note:* `gen_rand` as a traceful function also changes the underlying trace, but with `let*` we are only accessing the computed *value*. The new trace is passed on implicitly as before.

### Returning a value inside a traceful function

If we compute some value inside a traceful function, which we want to return to the caller of the function, we use `return value`.

**Example: Returning a Value inside a Traceful Function**

Consider the example from above, where we send a message and then store a state. Recall, that sending a message is just adding the message to the trace. If later, someone should receive this message, the receiver needs to know the timestamp of the message on the trace. Thus, the send function returns exactly this timestamp and our bigger function needs to also return the timestamp.

```
let* msg_ts = send_msg message in // send the message and store the returned timestamp
store_state alice state ;*        // do some other traceful actions
return msg_ts                     // return the message timestamp as final result of␣
↪the traceful function
```

### 4.2.2 The `option` Monad

Another thing, that we want to simplify is combining functions that may fail.

Consider as an example a function where we want to decrypt a cipher text, and then parse the resulting plain text to some abstract message format. Both of the steps (decryption and parsing) may fail and we want our overall function to succeed, only if both steps succeed.

Luckily, there is already a builtin type in F* capturing possible failure: the `option` type. So we can write our function as

```
match decrypt cipher_text with
| None -> None        // fail if decryption fails
| Some plaintext -> ( // decryption was successful
    match parse plaintext with
    | None -> None               // fail if parsing fails
    | Some abstract_plaintext -> // parsing was successful
        ... // do something with abstract_plaintext
)
```

This works perfectly fine, but is a lot of lines for just two actions!

To simplify the nested matches, we define the `option` monad. The corresponding side effect here would be failure, i.e., an optional action computes some value but may fail, with the intuition that the execution stops in the failure case.

Similarly to `;*` and `let*` for the `traceful` monad, we have `;?` and `let?` for composing functions that may fail and accessing return values of such functions.

**Example: `;?` and `let?` for the `option` monad**

Consider our example from before: a function that decrypts a cipher text, parses the resulting plain text and does something with the abstract plaintext. Each of the actions may fail and we want the overall function to succeed only if all actions succeed. We can write this as:

```
let? plaintext = decrypt cipher_text in // Try to decrypt the ciphertext.
    // The whole step fails, if decryption fails. If decryption is successful, the␣
↪result value is called plaintext.
let? abstract_plaintext = parse plaintext in // Try to parse the plaintext. If␣
↪successful the result value is called abstract_plaintext
some_function abstract_plaintext;?  // Do something with the abstract plaintext. This␣
↪step may fail (causing the whole function to fail).
Some abstract_plaintext  // if all steps succeeded, the final return value of the␣
↪function is the abstract plaintext
```

See how the previous two nested matches are now just two lines (the first two)!

### 4.2.3 The `traceful + option` Monad

Most of the actions in a protocol step will be actions that work with the trace *and* may fail. For example, in the last step of the Two-Message protocol, Alice checks whether she previously started a flow with the received nonce. This action needs to look at the trace, but may fail if she did not start a flow with this nonce.

So we have a combination of the previous two side effects. And indeed the combination of `traceful` and `option`, where we have `option` inside `traceful`, i.e., `traceful (option a)` is again a monad.

The intuition for sequentially composing two traceful + option actions is:

- the second action only gets executed if the first one is successful
- if the first action fails, the changes on the underlying trace of the first action are still captured

With this, we model that the individual steps may fail, but that all side-effects on the trace are recorded up until the point of failure.

Just like before we have `;*?` for sequential composition and `let*?` for accessing the return value of traceful + optional actions.

**Example: Composition in the `traceful + option` Monad**

Consider the following function in the traceful + option monad:

```
let f (x:int): traceful (option string) =
  if x < 2
  then (
    add_entry_ en;*?
    return (Some "added entry")
  )
  else return None
```

This function takes an `int` `x`, if `x` is less than 2, an entry `en` is added to the current trace and the string "added entry" is returned. If `x` is not less than 2, the function fails (and returns `None`).

We sequentially execute this function 3 times with different values for `x`:

```
let f3 = f 1;*? f 2;*? f 0
```

and see how the trace evolves (beginning with an empty trace) and what the final return value is.

1. We begin with an empty trace `tr0` and execute `f 1`. This returns an optional string `opt_str1` and a new trace `tr1`. Since 1 is less than 2, the entry `en` will be added to the trace `tr0` and `opt_str1` is `Some "added entry"`.
2. We then execute `f 2` with the new trace `tr1`. This again, returns an optional string `opt_str2` and a new trace `tr2`. Since 2 is not less than 2, `opt_str2` is `None` and the trace doesn't change.
3. Since the previous step returned `None`, the final `f 0` is not executed.

In total, after step 3 we have a return value `None` and a trace `[en]`.

To better understand the behaviour of the `traceful + option` monad, lets look at what happens if we switch the order of the three actions in the function `f3`: (the first row is the case we just looked at)

| f3 | return value | new trace |
|---|---|---|
| `f 1;*? f 2;*? f 0` | `None` | `[en]` |
| `f 0;*? f 1;*? f 2` | `None` | `[en; en]` |
| `f 2;*? f 1;*? f 0` | `None` | `[ ]` |
| `f 1;*? f 0` | `Some "added entry"` | `[en; en]` |

Looking at the first three rows, this highlights that monadic actions have *side effects*. Although the return value is the same for all three rows, the final trace is different.

TODO: where should the following box go?

**Remember**

The *last* action in a `traceful + option` function, must be

- a call to a `traceful + option` action or
- a `return` of `None` or `Some value`

### 4.2.4 Comparing `traceful + option` with `traceful`

The `traceful + option` monad can be considered a special case of the `traceful` monad: Every action in the `traceful + option` monad can be considered as an action in the plain `traceful` monad, if we ignore the knowledge that the return type is an option.

We already saw this in the function `f` in the previous example: The function uses the `return` of the `traceful` monad (in Lines 5 and 7) to return the computed value. Since, the return type of `f`, considered as a plain `traceful` action, is an option, we need to return `Some value` or `None`.

It is important to know which of the two monads we want, when composing `traceful + option` actions.

**Example: Comparing Composition of `traceful + option` actions with `traceful` actions**

To highlight the difference between sequential composition in the `traceful + option` monad with composition in the plain `traceful` monad, lets use `;*` instead of `;*?` in the function `f3` in the previous example:

```
let f3 = f 1;* f 2;* f 0
```

In the `traceful` monad, the composition just passes on the trace without looking at the return values of the function. Thus, the execution of `f3` does **not** stop when `f 2` fails (as in the previous example)! `f 2` doesn't change the trace and `f 0` is successfully executed.

The overall result of the new `f3` is `Some "added entry"` and the new trace is `[en; en]`.

Look back at the first line in the table in the previous example and compare the results when using `;*?`.

**Remember**

If we compose two `traceful + option` actions with `;*`, the second action gets executed in any case, even if the first action fails.

If we compose them instead with `;*?`, the second action is not executed when the first action fails.

TODO: add a link/pointer to the example file in the repo (examples/Basics/DY.Basics.Monads.fst)

### 4.2.5 Combining actions from different Monads

We have now seen three monads, and we looked at how to sequentially execute actions *within the same* monad. However, it is often the case that we need to combine actions from *different* monads.

**Example: The Three Monads in one Protocol Step**

Consider the following prototypical excerpt of a protocol step:

1. receive a message
2. parse the received message
3. send a new message

These actions are all in different monads:

1. receiving a message is a traceful action that may fail (if there is no message entry at the provided timestamp) (`traceful + option`)
2. parsing a message is an action that may fail, but does not need the trace (`option`)
3. sending a message is a traceful action that never fails (`traceful`)

It is possible to compose actions of different monads, but one has to be extra careful which composition to use and what the overall type of the function will be.

#### The Overall Type

Let's first think about the overall type of a function executing several actions from different monads.

Recall the main intuition: monadic actions are functions with side effects. If we compose several actions with different side effects, the overall function may have *all* of the individual side effects.

In our case, we have the three side effects of working with the trace, potential failure and "work with the trace and may fail". Observe, that the last one captures both side effects of the first two. From this, we intuitively see that whenever we compose actions from different monads, the overall function will be in the `traceful + option` monad, since we may have an effect on the trace and may fail.

Putting things a bit differently, this means, that if you want to write a function in the `option` monad, you *can not* use any `traceful` or `traceful + option` actions inside, since this would mean having more side effects than just the failure case from `option` specified in the type of your function. And similarly, if you are writing a non-optional `traceful` function, you can not use an `option` or `traceful + option` action inside, since the failure side effect is not covered by the specified type.

**Remember**

If you compse actions from different monads, you will end up in the `traceful + option` monad.

- Inside an `option` function, you *can not* call `traceful` or `traceful + option` actions.
- Inside a plain non-optional `traceful` function, you *can not* call `option` or `traceful + option` actions.

### How to Compose

Now that we know, we are in a `traceful + option` function, we have to think about how to call `option` and plain `traceful` actions inside our function.

#### `traceful` inside `traceful + option`

Plain non-optional `traceful` actions already have the "may modify trace" side effect, but are missing the "might fail" effect. Intuitively, a non-optional action, just does not fail and always produces a result and a new trace. We can thus use the plain `traceful` `;*`.

**Example: `traceful` action inside `traceful + option`**

In most of the protocol steps, we will send some message. This is a `traceful` action, since it appends the message to the trace, that returns the timestamp of the message in the new trace. The action never fails, since entries can always be added to the trace.

In the protocol step we can write

```
send_msg message;*
... // some other traceful + option actions
```

Or, if we want to use the returned timestamp

```
let* ts = send_msg message in
... // some other traceful + option actions that can use ts
```

Recall from `before` that the last action in a `traceful + option` function, needs to be a call to another `traceful + option` action (as in the example above), or a `return` of an `option`. So if we want to have a plain `traceful` action as the last step in our function, we need to have a `return` after that.

**Example: `traceful` action as *last* action inside `traceful + option`**

If sending a message is the last action in our protocol step, and we want to return the message timestamp, we need to write

```
let* ts = send_msg message in
return (Some ts)
```

#### `traceful + option` actions inside `traceful + option`

A special case of the above, are `traceful + option` actions inside the overall `traceful + option` function. You may wonder why we need to talk about this case here. Isn't this just combining actions within the *same* monad?

Well, we have seen `before` that we can use both `;*` and `;*?` to combine `traceful + option` actions. And it depends on whether we want the second action to be executed if the first one fails or not.

**Example: Using both `;*` and `;*?` for `traceful + option` actions**

We look again at the function `f` from `the previous example` and use a mix of `;*` and `;*?` to combine the calls for different arguments:

| f3 | return value | new trace |
|---|---|---|
| `f 1;*? f 2;* f 0` | `Some "added entry"` | `[en; en]` |
| `f 1;* f 3 ;* f 1 ;* f 2 ;*? f 0` | `None` | `[en; en]` |
| `f 1;* f 3 ;*? f 1 ;* f 2 ;*? f 0` | `None` | `[en]` |

### `option` inside `traceful + option`

An `option` action, already has the "may fail" effect, but is missing the "modifies trace" effect, since it doesn't even operate on a trace. Intuitively, an action not working with the trace can be seen as a `traceful` action that doesn't change the trace. That is, an `option` action can be seen as a `traceful` action that computes some value and passes the underlying trace on unchanged. This behavior can be realized with the `return` we previously introduced. `return` takes some value and attaches the current trace to it unchanged.

**Example: `option` action inside `traceful + option`**

When we receive a message in some protocol step, the message is in the wire format. Since we want to work with the abstract format, we need to parse the received message. Parsing may fail but does not need to look at the trace. Thus, parsing is an `option` action. The overall protocol step will be a `traceful + option` function.

Thus, we need to use `return` around the call to `parse` inside the protocol step:

```
let*? abstract_message = return (parse wire_message) in
```

TODO: add an exercise

**Remember**

To call an `option` action inside a `traceful + option` function, you need to use `return` around the call to the action.

### Summary

To summarize, it is possible to combine actions from different monads. If we do so, we will end up in the `traceful + option` monad. To combine these actions, we need to take special care of which `;` and `let` to use. The following gives an overview:

**Remember: Which `;` and `let` to use?**

If you want to call an action `act : m ret` inside a function `f : m' ret'`, where `m` and `m'` are two of the three monads we have seen, and `ret` and `ret'` some return types, you can use the following table to choose the right composition and accessing return values:

| `m'` (the monad of `f`) | `m` (the monad of `act`) | `;` | `let` | comment |
|---|---|---|---|---|
| `option` | `option` | `;?` | `let?` | |
| `traceful` | `traceful` | `;*` | `let*` | |
| `traceful + option` | `traceful + option` | `;*` or `;*?` | `let*` or `let*?` | depending on wanted behavior in error case: should the rest be executed if the first action fails or not? |
| | `option` | `;*?` | `let*?` | *must* use `return` around call to action |
| | `traceful` and `ret'` is not `option` | `;*` | `let*` | if *last* action in `f`, a `return` must follow |

Plain `let` (i.e., non-monadic actions) can be used inside any monadic function.

We now have all pre-requisites to start our first protocol model in DY*.

## 4.3 A Model of the Two-Message Protocol

We build the DY* model of the Two-Message protocol following the *local-view description* step by step.

### 4.3.1 Setup

Create a new folder `TwoMessageP` and add the following `Makefile`:

```
TUTORIAL_HOME ?= ../..

EXAMPLE_DIRS = path_to/TwoMessageP

include $(TUTORIAL_HOME)/Makefile
```

If your run `make` in the `TwoMessageP` folder now, it will verify all of core DY*. This produces a lot of output on the terminal (including some warnings) and takes quite some time (around 10 Minutes)! You only have to do this once though, so you may as well start it now and let it run.

It is best practice to separate the definition of the abstract message and state types from the protocol steps. We create two files `DY.TwoMessage.Data.fst` and `DY.TwoMessage.Protocol.fst` in the `TwoMessageP` folder. Both of them are F* modules and as such must begin with

```
module name_of_file_without_fst
```

To import functions from core DY*, we need to open the corresponding modules in both:

```
open Comparse
open DY.Core
open DY.Lib
```

If the environment variables and the Makefile are setup correctly, you should be able to (interactively) check both of the files successfully. If the previous `make` has finished and you do `make` again, it should be much faster and you should see a few lines related to the two files in the folder.

### 4.3.2 The abstract message and state types

We define the types for the abstract message and state format in the `DY.TwoMessage.Data` file.

### Messages

When we previously introduced the abstract message format and looked at the example for the Two-Message protocol, we already used valid F* syntax. So we can just copy the two record type definitions for `ping_t` and `ack_t` from the `previous example`.

Now that we have the types for the two messages separately, we want to collect them in a single type for messages. We do this, by defining a new type `message_t` and use constructors for the two messages:

```
type message_t =
  | Ping: (ping:ping_t) -> message_t
  | Ack:  (ack:ack_t) -> message_t
```

This fully defines our abstract message format. Whenever we have a `msg` of type `message_t`, we know from the constructor whether it is a Ping or Ack message and we can access the corresponding fields from the `ping_t` or `ack_t` record.

**Example: Recap – Instantiating and Accessing Values of Record Types**

A concrete instance of a Ping message with content "Alice" and $n$, can be specified as

```
let msg : message_t = Ping {alice = "Alice"; n_a = n}
```

Given this `msg`, we can access say the `alice` field value with:

```
let al = (Ping?.ping msg).alice
```

We now need to deal with parsing and serializing of our abstract message format from/to the wire format. Luckily, we don't have to write parsing and serializing by hand, since the tool Comparse can construct those automatically from the type definitions.

To invoke Comparse for a given type, we have to call

```
%splice [ps_(name_of_type)] (gen_parser (`name_of_type))
```

In our case, we need:

```
%splice [ps_ping_t] (gen_parser (`ping_t))

%splice [ps_ack_t] (gen_parser (`ack_t))

%splice [ps_message_t] (gen_parser (`message_t))
```

This generates parsing and serializing functions for each of our abstract message types.

However, this does not quite work yet. We need to guide Comparse a bit. Since Comparse is so general, we need to add some annotations to our types. We need to add `[@@ with_bytes bytes]` right before the type definitions, i.e.,

```
[@@ with_bytes bytes]
type ping_t = {
  alice: principal;
  n_a : bytes;
}
```

We need to do that also for `ack_t` and `message_t`.

Once you added all annotations, the splices should verify. Verifying these lines takes a while (up to 15 Seconds)! This is one of the reasons to separate them from other functions that you may want to re-check frequently (for example protocol steps).

**Common Mistake: Forgetting `[@@ with_bytes bytes]`**

If you try `%splice [ps_ping_t] (gen_parser (`ping_t))`, but get an error of the form

```
- 'tc' failed
- 'tcc' failed
- Cannot type (3) DY.Core.Bytes.Type.ps_bytes in context ([]). Error = ( - Name
  "DY.Core.Bytes.Type.ps_bytes" not found )
```

you probably forgot to add the annotation `[@@ with_bytes bytes]` right before the type you are splicing (here `ping_t`).

**Common Mistake: Not using `ps_name_of_type` in the first argument of `splice`**

If you get an error of the form

```
Splice declared the name DY.TwoMessageP.Data.something_else_than_ps_ack_t but it was␣
↪not defined.
Those defined were: [DY.TwoMessageP.Data.ps_ack_t]
```

you probably didn't use `ps_ack_t` in the brackets of the splice.

The term in this brackets really needs to be `ps_` followed by the name of the type (here `ack_t`)!

That is, `%splice [something_else_than_ps_ack_t] (gen_parser (`ack_t))` will produce the above error, while `%splice [ps_ack_t] (gen_parser (`ack_t))` will not.

The final step for serializing and parsing is to declare our message type `message_t` an instance of Comparse's `parseable_serializable` class, like this:

```
instance parseable_serializeable_bytes_message_t: parseable_serializeable bytes message_
↪t =
  mk_parseable_serializeable ps_message_t
```

We don't need to understand the details of this line. The important part is, that now we can call `parse` and `serialize` for instances of our message type.

**Example: `serialize` and `parse` for `message_t`**

With `let msg : message_t = Ping {alice = "Alice"; n_a = empty}` from above, we can write

```
let _ =
  let msg_wire : bytes = serialize message_t msg in // serialize the abstract format␣
↪to the wire format
  let msg_ = parse message_t msg_wire in  // parse from the wire format to an␣
↪abstract message
```

Note that we have to tell `serialize` and `parse` what the abstract type is that we consider (here `message_t`).

We don't need to care about *how* the abstract messages are translated to the wire format, i.e., how `msg_wire` looks like.

We just use the following properties that Comparse provides automatically:

```
let parse_serialize_inv_lemma x =
  parse message_t (serialize message_t x) == Some x

let serialize_parse_inv_lemma buff =
  match parse message_t buf with
  | Some x -> serialize message_t x == buf
  | None -> True
```

**Discussion**

These inverse properties of parsing and serializing are quite strong and may not be true for some real-world protocols! There might be different abstract messages that have the same serialization. Or the other way around: a wire format message could be parsed to two different abstract formats. There are examples of format confusion attacks for several protocols.

Thus, enforcing the inverse properties "hides" some attacks that can not be found when using Comparse for generating the parser and serializer of a type.

Your `DY.TwoMessageP.Data.fst` file should now look like this:

```
module DY.TwoMessageP.Data

open Comparse
open DY.Core
open DY.Lib

[@@ with_bytes bytes]
type ping_t = {
  alice: principal;
  n_a : bytes;
}

[@@ with_bytes bytes]
type ack_t = {
  n_a : bytes;
}

[@@ with_bytes bytes]
type message_t =
  | Ping: (ping:ping_t) -> message_t
  | Ack:  (ack:ack_t) -> message_t

%splice [ps_ping_t] (gen_parser (`ping_t))
%splice [ps_ack_t] (gen_parser (`ack_t))
%splice [ps_message_t] (gen_parser (`message_t))

instance parseable_serializeable_bytes_message_t: parseable_serializeable bytes message_
↪t =
  mk_parseable_serializeable ps_message_t
```

We defined types for the abstract formats for Ping and Ack messages. We then collected them in an overall `message_t` type. Finally, we used Comparse to generate parsers and serializers for our types, so that we can call `parse` and `serialize` for values of type `message_t`.

### States

We do the exact same thing for our `abstract state formats`.

**Exercise: Define the abstract state format and parsing and serializing for them**

1. Copy the record types for the single abstract states.

   **Show/Hide Answer**

```
type sent_ping_t = {
  bob : principal;
  n_a : bytes;
}

type sent_ack_t = {
  alice: principal;
  n_a : bytes;
}

type received_ack_t = {
  bob : principal;
  n_a : bytes;
}
```

2. Collect the types for the single states into an overall `state_t` type.

   **Show/Hide Answer**

```
type state_t =
  | SentPing: (ping:sent_ping_t) -> state_t
  | SentAck: (ack:sent_ack_t) -> state_t
  | ReceivedAck: (rack:received_ack_t) -> state_t
```

3. Use Comparse to generate a parser and serializer for `state_t`.

   **Show/Hide Answer**

```
%splice [ps_sent_ping_t] (gen_parser (`sent_ping_t))
%splice [ps_sent_ack_t] (gen_parser (`sent_ack_t))
%splice [ps_received_ack_t] (gen_parser (`received_ack_t))
%splice [ps_state_t] (gen_parser (`state_t))
```

   Don't forget to add `[@@ with_bytes bytes]` for every type!

4. Make `state_t` an instance of Comparse's `parseable_serializable` class.

   **Show/Hide Answer**

```
instance parseable_serializeable_bytes_state_t: parseable_serializeable bytes␣
↪state_t =
  mk_parseable_serializeable ps_state_t
```

There is one final technicality that we need to do for the state type. We need to make it an instance of DY*'s `local_state` like this:

```
instance local_state_state_t: local_state state_t = {
  tag = "TwoMessage.State";
```

(continues on next page)

(continued from previous page)

```
  format = parseable_serializeable_bytes_state_t;
}
```

This ensures that our protocol-level states do not interfere with any state internal to DY*.

#### Summary

We now have our abstract message and state types and can translate those to and from the wire format. This is all that goes into the `Data` module.

**How-To: Define abstract message and state formats**

- The abstract formats for messages and states go into the `Data` module.
- For each message and state in the protocol, define its own type as a record.
- Collect the individual record types for messages and states in an overall type `message_t` and `state_t` using constructors for each of the cases.
- Use Comparse to generate a parser and serializer for the types, with `%splice [ps_name_of_type] (gen_parser (`name_of_type))` and `[@@ with_bytes bytes]` annotations.
- Define `parseable_serializable` instances for `message_t` and `state_t`.
- Define `local_state` instance for `state_t`.

### 4.3.3 The protocol steps

We will now implement the protocol steps in the `DY.TwoMessage.Protocol` module.

First, we need to import some libraries that we need later. From above, we already have

```
open Comparse
open DY.Core
open DY.Lib
```

we add two more DY* libraries from the tutorial repo, that offer a simplified interface to some of the core DY* functions:

```
open DY.Simplified
open DY.Extend
```

These are imports for DY* functionality, but we also want to use the abstract message and state types, we just defined. So we finally need to import

```
open DY.TwoMessage.Data
```

Our model of the Two-Message Protocol, will consist of 3 functions, one for each step of the *local-view description*.

### The first step: Sending a Ping

Recall what happens in the first step of the protocol, where Alice sends a Ping message:

1. Alice generates a new nonce $n_A$
2. Alice sends the Ping message containing the nonce $n_A$ together with her name.
3. She stores the nonce $n_A$ and Bob's name in a new session.

### The type

Let's first think about the type of this protocol step. Since we will send a message and store a new state, the step is a `traceful` function. Sending a message and storing a new state are both actions that will always succeed, so we have a plain non-optional `traceful` function.

The input arguments are the names of Alice and Bob, where Alice is the acting party and Bob is the intended receiver of the message.

The return values are the session ID of the new session of Alice, where she stored the nonce, so that we know which of Alice's sessions is the one related to this run of the protocol and the timestamp of the Ping message, so that we know where Bob can read the Ping on the trace.

In total, we have:

```
val send_ping:
  (alice: principal) -> (bob: principal) ->
  traceful (state_id & timestamp)
```

### Generating the nonce

To generate the nonce $n_A$, we use the function `gen_rand` from the `DY.Simplified` library. This is a `traceful` action, so we need to use `let*` to give a name to the nonce:

```
let* n_a = gen_rand in
```

### Sending the message

We build the abstract Ping message with

```
let ping = Ping {alice = alice; n_a = n_a} in
```

Note that this is a non-monadic action, i.e., an action without any side effects. We are just defining a concrete instance of an abstract message. Hence, we use the plain `let` here. Recall, that plain `let` s can be used inside any monad.

Before sending the Ping, we need to translate it to the wire format. That is, we need to serialize it:

```
let ping_wire = serialize message_t ping in
```

Again, serializing is a non-monadic action, so we have the plain `let`.

Observe that we need to tell the `serialize` function the type of the second argument (here `message_t`), so that it knows which format to use.

Now, we can send the serialized Ping message with the `send_msg` function from `DY.Core`. This function is a `traceful` function that adds the message to the trace and returns the timestamp of the message on the trace. This time, we have to use `let*` to give the returned timestamp a name (`traceful` action inside a `traceful` function):

```
let* msg_ts = send_msg ping_wire in
```

The message is now sent and we can turn to storing the new state.

### Storing the state

As for the message, we first define the abstract state with

```
let ping_state = SentPing {bob = bob; n_a = n_a} in
```

We can then use the `start_new_session` function from the simplified `DY.Simplified` library, which returns the sessoin ID of the new session that was created. This is a `traceful` action, hence we use `let*` again.

```
let* sid = start_new_session alice ping_state in
```

Note, that in contrast to messages, we do not need to serialize the abstract state first! This is handled internally by the `start_new_session` function which is very convenient.

That's it for storing the state.

### Returning

We performed all actions (generating the nonce, sending the Ping message and storing a new state) and can now finish the protocol step by returning the new session id and the message timestamp:

```
return (sid, msg_ts)
```

### Summary

In total, the function looks like this:

```
val send_ping:
  (alice: principal) -> (bob: principal) ->
  traceful (state_id & timestamp)
let send_ping alice bob =
  let* n_a = gen_rand in

  let ping = Ping {alice = alice; n_a = n_a} in
  let ping_wire = serialize message_t ping in
  let* msg_ts = send_msg ping_wire in

  let ping_state = SentPing {bob = bob; n_a = n_a} in
  let* sid = start_new_session alice ping_state in

  return (sid, msg_ts)
```

### The second step: Replying with an Ack

In the second protocol step, Bob receives a Ping message, sends back the received nonce and stores a new state for the session with Alice and the none received in the message.

### The Type

We begin again with the type of this second step. We are now reading from the trace, sending a message and storing a new state. So this step is also a `traceful` function. But now, reading the Ping message may fail (either when receiving, if there is no message entry at the given timestamp or when parsing the wire format message to a Ping) and so the whole step mail fail and we end up with a `traceful` + `option` function.

The input arguments are the name of Bob (the receiving and replying participant) and the timestamp of the Ping message.

The return values are the same as for the first protocol step: the session ID of the new session of Bob, where he stores the received nonce and name for this protocol run and the timestamp of his reply (the Ack).

Thus, we have the type:

```
val receive_ping_and_send_ack:
  (bob:principal) -> (ping_ts:timestamp) ->
  traceful (option (state_id & timestamp))
```

### Receiving the Ping

To read the message from the trace, i.e., receiving a message, we call the `recv_msg` function from `DY.Core`. It takes as an argument the timestamp of the message and returns the content of the message in wire format. This can fail, if the entry at the provided timestamp is not a message entry (for example, if it is a state entry, or a generate nonce entry). `recv_msg` is thus a `traceful` + `option` function. Since we want the overall protocol step to fail, if receiving the message fails, we use `let*?` to store the read message content:

```
let*? msg = recv_msg msg_ts in
```

If receiving is successful, `msg` contains the content of the message and the step continues. We now have to translate the wire format `msg` to our abstract Ping type. Translating from wire to abstract format is parsing and we would like to call

```
parse message_t msg
```

Parsing is an action that might fail, but does not need to look at or change the trace. Hence, parsing is an `option` action. Now recall from the `monad introduction` that if we want to use an `option` action inside a `traceful` + `option` action, we need to use a `return` around the call. And again, we want our overall step to fail, if parsing fails, so we use `let*?`:

```
let*? png = return (parse message_t msg) in
```

If parsing is successful, we now have the abstract format of the message content stored at the input argument timestamp. Now, we need to check that the received message is indeed a Ping message (and not an Ack for example). If it is not a Ping message, the whole step should fail.

For this check, we use the `guard_tr` function from `DY.Core`. This function takes a `bool` as argument and has the behavior: if the bool is `true`, the execution continues with an unchanged trace, if it is `false` the execution stops.

With this, we can write:

```
guard_tr (Ping? png);*?
```

This checks, if the received abstract message is a Ping or not and stops the execution if not.

Now, we know that the received message is indeed a Ping and we can finally access the values that we need:

```
let Ping png = png in
let alice = png.alice in
let n_a = png.n_a in
```

Note: The first line above just removes the `Ping` constructor. The received and parsed `png` is of the general `message_t` type, but to access the field values of the `ping_t` record, we need to remove the `Ping` constructor first.

This concludes receiving of the message and reading its content.

### Sending the Ack

We can now build the reply by defining the concrete instance of the `ack_t`:

```
let ack = Ack {n_a} in
```

To send the reply, we first need to serialize it to the wire format and can then use the `send_msg` function, we already saw in the first protocol step. This time, we do both serializing and sending together:

```
let* ack_ts = send_msg (serialize message_t ack) in
```

Remember that sending a message is a `traceful` action that does not fail, hence we have to use `let*` here.

### Storing the State

The remaining part of the second protocol step is storing a new state for Bob. This works the same as in the first protocol step. We first define the content of the new state in the abstract format:

```
let ack_state = SentAck {alice; n_a} in
```

and then start a new session for Bob with this new content:

```
let* sess_id = start_new_session bob ack_state in
```

Recall, we don't have to serialize the state content, as this is done implicitly inside the `start_new_session` function. Again, this is a `traceful` function that does not fail, hence the `let*`.

### Returning

Finally, we have to return the session ID of the new session and the timestamp of the Ack. Remember, that we are now in a `traceful` + `option` function, hence we need to return an `option` (in contrast to the first protocol step):

```
return (Some (sess_id, ack_ts))
```

### Summary

Collecting everything from above, the second protocol step looks like this:

```
val receive_ping_and_send_ack:
  (bob: principal) -> (ping_ts: timestamp) ->
  traceful (option (state_id & timestamp))
let receive_ping_and_send_ack bob msg_ts =
  let*? msg = recv_msg msg_ts in
  let*? png = return (parse message_t msg) in
  guard_tr (Ping? png);*?

  let Ping png = png in
  let alice = png.alice in
  let n_a = png.n_a in

  let ack = Ack {n_a} in
  let* ack_ts = send_msg (serialize message_t ack) in

  let ack_state = SentAck {alice; n_a} in
  let* sess_id = start_new_session bob ack_state in

  return (Some (sess_id, ack_ts))
```

### The third step: Receiving an Ack

In the final protocol step, Alice receives an Acknowledgment from Bob and checks whether she previously started a run with Bob and the nonce received in the Ack. If this is the case, she stores that this run is now completed.

### The Type

This step reads the Ack message from the trace and looks for a previous state of Alice on the trace and updates that state. The step is thus a `traceful` function. As in the previous step, reading (and parsing) the Ack message may fail as well as finding a previous state of Alice. We thus have a `traceful` + `option` function.

The input arguments are the name of the acting participant, i.e., Alice and the timestamp of the acknowledgment.

The return value is the session ID of the session that Alice now considers as completed.

The type is:

```
val receive_ack:
  (alice: principal) -> (ack_ts: timestamp) ->
  traceful (option state_id)
```

### Receiving the Ack

Receiving and parsing of the Ack is the same as for the Ping message in the previous step.

We first call the `recv_msg` function with the provided timestamp of the Ack. Recall, this is a `traceful + option` action that might fail, if the trace entry at this timestamp is not a message entry. We thus have to use `let*?`

```
let*? msg = recv_msg ack_ts in
```

If this action is successful, we now have the Ack in the wire format. To translate it to our abstract format, we call `parse`. Since this is an `option` action inside a `traceful + option` function, we need to wrap the call in a `return` and use `let*?`:

```
let*? ack = return (parse message_t msg) in
```

If this action is successful, we have the abstract message that was on the trace at the provided timestamp. We now need to check that this message is indeed an Ack. For this we use again the `guard_tr` function:

```
guard_tr (Ack? ack);*?
```

Reading the Ack from the trace was successful and we can now access the data with

```
let Ack ack = ack in
let n_a = ack.n_a in
```

Note again, that the first line just removes the `Ack` constructor to then access the `n_a` record field of the `ack_t` type.

### Searching a previous state

At this point, Alice received a nonce $n_A$ as an acknowledgment. She now wants to check, whether she actually started a run with this nonce.

That is, we are looking for a previous state of Alice that is a `SentPing` state and contains the same nonce $n_A$.

For this, we use the `lookup_state` function from `DY.Extend`:

```
let*? (sid, st) = lookup_state #state_t alice
  (fun st ->
        SentPing? st
     && (SentPing?.ping st).n_a = n_a
  ) in
```

This function takes three arguments:

- the implicit abstract state type (here `state_t`)
- the participant for which we want to find a state (here `alice`)
- a property of an abstract state `propt: state_t -> bool`

  Here the property checks that the state is a `SentPing` state and the stored nonce in the state is the same as the received nonce $n_A$ from the Ack message.

The lookup function returns the session ID of the found state and the content, i.e. the abstract state. The action may fail, if there is no state of the principal satisfying the property.

**Discussion: How lookup works**

The `lookup_state` function looks for the first state entry on the trace (starting from the back/most recent time) that satisfies the property. It does *not* check that this state entry is the most recent one in the session. I.e., it could be the case, that the returned state is not the current state of the principal. In our case here that means, that we do not check whether Alice already completed the run before.

For the examples in this tutorial, this implementation of the lookup function is sufficient. A future feature in DY* will change the behavior of the lookup to include the check to look only at the most recent states of sessions.

**Discussion: Why use lookup?**

From the analysis of cryptographic protocols you may be used to the attacker specifying the concrete session in follow-up steps. In our example, we could have added an additional argument to the `receive_ack` step for the session that Alice should be in. I.e., she would check whether the received nonce corresponds to the nonce stored in that specific session and then finish this session.

Instead, we use the state lookup function here which is closer to how web servers work. They look at the content of received messages and try to find the corresponding session from that. For example by looking at some session ID provided in the message.

Both models cover a range of session mixup attacks. If the attacker can provide the session, he can inject data in possibly unrelated sessions. With the lookup, we also get session mixup since the key we use for lookup might not be unique across sessions.

Both models can be implemented in DY*.

Now that we have the previous state of Alice, we can access the fields and see who was the other participant of that run

```
let bob = (SentPing?.ping st).bob in
```

Unfortunately, this does not work immediately as we would expect. We need to use another `guard_tr` first.

```
guard_tr (SentPing? st);*?
```

This guard should always be satisfied, since the lookup function returns a state satisfying the property, which already includes that the state is a `SentPing` state. So it doesn't change the function, but it is needed for a technical reason for DY* to argue that the found state is a `SentPing` state.

**The technical reason**

In the bind of the `traceful` + `option` monad, the second action can't make use of the fact, that it is executed only if the first action was successful. It turns out to be technically very difficult to express this behavior for the second action in the type definition of the bind.

We have now found that Alice previously started a run with the nonce received in the Ack.

### Setting the new state

The next action is finishing the run. We do this by updating the previous session of Alice to a `ReceivedAck` state.

Recall from the introduction of *States* and *The Global Trace* that a (full) state of a principal consists of several sessions and that state entries on the trace contain only a single state and not a complete session or full state of a principal. The content of a state are stored together with the name of the principal and the session ID on the trace. We may have several state entries for the same principal and session ID. In this case the intended meaning is, that the most recent of those entries is the current state of this session.

Updating the state of a session, is thus just adding a new state entry to the trace with the new content for the same principal and session ID. This can be done with the `set_state` function:

```
set_state alice sid (ReceivedAck {bob; n_a});*
```

It is a `traceful` action that does not fail, it takes as arguments the principal (`alice`), the session ID (`sid`) and the content of the state in its abstract format (the `ReceivedAck` state)

### Returning

The protocol step returns the session ID of the session of Alice that has now finished.

```
return (Some sid)
```

### Summary

The final protocol step looks like this:

```
val receive_ack:
  (alice: principal) -> (ack_ts: timestamp) ->
  traceful (option state_id)
let receive_ack alice ack_ts =
  let*? ack = recv_msg ack_ts in
  let*? ack = return (parse message_t ack) in
  guard_tr (Ack? ack);*?

  let Ack ack = ack in
  let n_a = ack.n_a in

  let*? (sid, st) = lookup_state #state_t alice
    (fun st ->
          SentPing? st
      && (SentPing?.ping st).n_a = n_a
    ) in
  guard_tr(SentPing? st);*?
  let bob = (SentPing?.ping st).bob in

  set_state alice sid (ReceivedAck {bob; n_a});*

  return (Some sid)
```

This concludes the model of the Two-message protocol. We defined the abstract message and state types in `DY.TwoMessage.Data` and the three protocol steps as functions in `DY.TwoMessage.Protocol`.

You should be able to run `make` now in the example folder.

### 4.3.4 An Example Protocol Run

Now that we have our model of the Two-Message protocol, we need to check that we modeled the right behavior. We do this by implementing an example run of the protocol using the protocol functions from the model. We can then print the trace after this run and see whether the run finishes successfully and whether the trace looks as we expect.

Recall from *before* that the trace after one successful run of the Two-Message protocol should look like this:

```
1. Generate nonce n_A
2. Message: Ping (Alice, n_A)
3. Session 0 of Alice: SentPing n_A to Bob
4. Message: Ack n_A
5. Session 0 of Bob: SentAck n_A to Alice
6. Session 0 of Alice: ReceivedAck n_A from Bob
```

#### Setup

We implement the example run in a new module `DY.TwoMessage.Run` in the same `TwoMessageP` folder. We begin the module with the usual imports of the DY* libraries and of course our protocol model:

```
open DY.Core
open DY.Lib
open DY.Simplified

open DY.TwoMessage.Protocol
```

As a second setup, we add a new target to our `Makefile` so that we can execute the extracted code and print the trace. Add the following at the end of the `Makefile` in the `TwoMessageP` folder:

```
test:
	cd $(TUTORIAL_HOME)/obj; $(FSTAR_EXE) --ocamlenv ocamlbuild -use-ocamlfind -pkg␣
↪batteries -pkg fstar.lib DY_TwoMessage_Run.native
	$(TUTORIAL_HOME)/obj/DY_TwoMessage_Run.native
```

If you now run `make` followed by `make test`, the second command should produce some warnings on the terminal and end with the following:

```
Finished, 4 targets (0 cached) in 00:00:00.
../../obj/DY_TwoMessage_Run.native
```

You are now set up to start implementing the example protocol run in the new `DY.TwoMessage.Run` module. This will consist of three parts:

1. Calling the protocol steps.
2. Printing the trace after the run.
3. Executing the example run.

### The example run

The example run, will be a function where we call the protocol steps in the expected order. Since the individual steps are functions in the `traceful + option` monad, our overall run function will also be in the `traceful + option` monad. The run does not take any input arguments and does not return any value. The type of the run function is thus

```
val run: unit -> traceful (option unit)
```

The first step of the protocol is the `send_ping` step. This step takes as arguments the names of the two participants, i.e., Alice and Bob. Since we need these names later on again, it is useful to define them in two variables:

```
let alice = "Alice" in
let bob = "Bob" in
```

We can now call the `send_ping` step with `alice` and `bob`. This is a `traceful` action that returns the new session ID of Alice for this new run and the timestamp of the Ping message on the trace. For the second step, we need the timestamp of the Ping, so we save the computed values of the `send_ping` step with `let*`:

```
let* (alice_sid, ping_ts) = send_ping alice bob in
```

The second protocol step (`receive_ping_and_send_ack`) takes as arguments the name of Bob (i.e., the variable `bob`) and the timestamp of the Ping that the first step just computed (`ping_ts`). The second step is a `traceful + option` action and we want the overall run to fail, if this step fails, so we use `let*?` to store the computed values:

```
let*? (bob_sid, ack_ts) = receive_ping_and_send_ack bob ping_ts in
```

The final step takes as argument the name of Alice (stored in `alice`) and the timestamp of the Ack. This last step is also a `traceful + option` action and we want the run to fail, if this step fails. Since we don't want to continue our run after this step, we don't need to save the computed values and can just use:

```
receive_ack alice ack_ts;*?
```

This concludes the implementation of the example protocol run. We called the protocol steps in the expected order, passing on the arguments (in particular the message timestamps) from the previous steps.

### Printing the Trace

Inside the `run` function, we want to print the trace, after the run.

Printing in F* works with `IO.debug_print_string` which takes a string as argument that is then printed. As you noticed before, the `make test` produced some lines of output, even though we didn't have any run yet. It is thus useful, to print some recognizable string before the actual output starts. For example like this:

```
let _ = IO.debug_print_string "************** Trace *************\n" in
```

To print the actual trace, we use the `trace_to_string` function from `DY.Lib`. This function takes two arguments: some printer functions and the trace to be printed. It returns a string. For now, we can just use the default printers `default_trace_to_string_printers`.

But how do we get the trace after the example run? There is the `traceful` function `get_trace` in `DY.Core` that returns the current trace. We can thus access the trace after the example run, by calling `get_trace` after the calls to the protocol steps:

```
let* tr = get_trace in
```

Now that we have the trace, we can call `trace_to_string` and print the resulting string:

```
let _ = IO.debug_print_string (
   trace_to_string default_trace_to_string_printers tr
 ) in
```

This concludes printing of the trace.

However, the `run` function has return type `traceful (option unit)`, so we need to return something in the end. In our case, we can just go with

```
return (Some ())
```

The overall `run` function looks like this:

```
val run:
  unit -> traceful (option unit)
let run () =
  let alice = "Alice" in
  let bob = "Bob" in

  let* (alice_sid, ping_ts) = send_ping alice bob in
  let*? (bob_sid, ack_ts) =
    receive_ping_and_send_ack bob ping_ts in
  receive_ack alice ack_ts;*?


  let* tr = get_trace in
  let _ = IO.debug_print_string "************* Trace *************\n" in
  let _ = IO.debug_print_string (
    trace_to_string default_trace_to_string_printers tr
    ) in

  return (Some ())
```

You can run `make` followed by `make test` now. But you will not see any output from the `run` function, only some warnings and output from the extraction process.

### Executing the Run

As a final step, we need to call our `run` function, when the module is executed. We achieve this by adding

```
let _ = run () empty_trace
```

at the end of the module.

If you run `make` followed by `make test` now, you will see the `*** Trace ***` string followed by the trace after the run. The trace should look like this:

```
{"TraceID": 0, "Type": "Nonce", "Usage": {"Type": "NoUsage"}}
{"TraceID": 1, "Type": "Message", "Content": "[Alice, [Nonce #0,]]"}
{"TraceID": 2, "Type": "Session", "SessionID": 0, "Principal": "Alice", "Tag":
↪"TwoMessage.State", "Content": "[Bob, [Nonce #0,]]"}
{"TraceID": 3, "Type": "Message", "Content": " [Nonce #0,]"}
{"TraceID": 4, "Type": "Session", "SessionID": 0, "Principal": "Bob", "Tag": "TwoMessage.
↪State", "Content": "[Alice, [Nonce #0,]]"}
```

(continues on next page)

(continued from previous page)

```
{"TraceID": 5, "Type": "Session", "SessionID": 0, "Principal": "Alice", "Tag":
↪"TwoMessage.State", "Content": "[Bob, [Nonce #0,]]"}
```

### Pretty Printing

We can print the content of the trace entries nicer, by defining our own printer functions for our message and state types. Download the `DY.TwoMessage.Run.Printing.fst` file from the Tutorial Repo on GitHub. Import the module in the `DY.TwoMessage.Run` module and change the default printers to `get_trace_to_string_printers`.

The printing should now look like:

```
let _ = IO.debug_print_string (
   trace_to_string get_trace_to_string_printers tr
 ) in
```

Run `make` followed by `make test` again. The trace should now look like this:

```
{"TraceID": 0, "Type": "Nonce", "Usage": {"Type": "NoUsage"}}
{"TraceID": 1, "Type": "Message", "Content": "Ping [name = (Nonce #0), n_a = (Alice)]"}
{"TraceID": 2, "Type": "Session", "SessionID": 0, "Principal": "Alice", "Tag":
↪"TwoMessage.State", "Content": "SentPing [n_a = (Nonce #0), to = (Bob)]"}
{"TraceID": 3, "Type": "Message", "Content": "Ack [n_a = (Nonce #0)]"}
{"TraceID": 4, "Type": "Session", "SessionID": 0, "Principal": "Bob", "Tag": "TwoMessage.
↪State", "Content": "SentAck [n_a = (Nonce #0), to = (Alice)]"}
{"TraceID": 5, "Type": "Session", "SessionID": 0, "Principal": "Alice", "Tag":
↪"TwoMessage.State", "Content": "ReceivedAck [n_a = (Nonce #0), from = (Bob)]"}
```

Which is much closer to our *previous* intuitive description of the trace.

## 4.4 How To: Modelling a Protocol in DY*

Before we continue to model the next protocol, let's summarize how we build a model of a protocol in DY* using our model of the Two-Message protocol as an example.

**How-To: Modelling a Protocol in DY***

*Technical Setup:*

- create a new folder
- add a `Makefile`

```
TUTORIAL_HOME ?= ../..

EXAMPLE_DIRS = path_to/TwoMessageP

include $(TUTORIAL_HOME)/Makefile
```

- create two modules: `Data.fst` and `Protocol.fst`

```
module DY.TwoMessage.Data

open Comparse
open DY.Core
open DY.Lib
```

```
module DY.TwoMessage.Protocol

open Comparse
open DY.Core
open DY.Lib

open DY.Simplified
open DY.Extend

open DY.TwoMessage.Data
```

*Define abstract message and state formats:*

- The abstract formats for messages and states go into the `Data` module.
- For each message and state in the protocol, define its own type as a record.

```
type ping_t = {
  alice: principal;
  n_a : bytes;
}

type ack_t = {
  n_a : bytes;
}
type sent_ping_t = {
  bob : principal;
  n_a : bytes;
}

type sent_ack_t = {
  alice: principal;
  n_a : bytes;
}

type received_ack_t = {
  bob : principal;
  n_a : bytes;
}
```

- Collect the individual record types for messages and states in an overall type `message_t` and `state_t` using constructors for each of the cases.

```
type message_t =
  | Ping: (ping:ping_t) -> message_t
  | Ack:  (ack:ack_t) -> message_t
```

```
type state_t =
  | SentPing: (ping:sent_ping_t) -> state_t
  | SentAck: (ack:sent_ack_t) -> state_t
  | ReceivedAck: (rack:received_ack_t) -> state_t
```

- Use Comparse to generate a parser and serializer for the types, with `%splice [ps_name_of_type] (gen_parser (`name_of_type))` and `[@@ with_bytes bytes]` annotations.

```
%splice [ps_ping_t] (gen_parser (`ping_t))
%splice [ps_ack_t] (gen_parser (`ack_t))
%splice [ps_message_t] (gen_parser (`message_t))
```

```
[@@ with_bytes bytes]
type ping_t = {
  alice: principal;
  n_a : bytes;
}
```

- Define `parseable_serializable` instances for `message_t` and `state_t`.

```
instance parseable_serializeable_bytes_message_t: parseable_serializeable bytes␣
↪message_t =
  mk_parseable_serializeable ps_message_t
```

```
instance parseable_serializeable_bytes_state_t: parseable_serializeable bytes␣
↪state_t =
  mk_parseable_serializeable ps_state_t
```

- Define `local_state` instance for `state_t`.

```
instance local_state_state: local_state state_t = {
  tag = "P.State";
  format = parseable_serializeable_bytes_state_t;
}
```

*Implement the protocol steps:*

- The protocol steps go into the `Protocol` module.
- For each step think about the *type* first:
  - The *monad*:
    - Every step will be in the `traceful` monad, since it works with the trace.
    - The step is in the `traceful` + `option` monad, if it may fail. (For example when reading a message from the trace, parsing a message to an expected format, looking up a particular state, etc.)
    - Most steps will probably be in the `traceful` + `option` monad. (A typical exception is the very first step of the protocol.)
  - Typical *input arguments* are:
    - the active participant in this step (of type `principal`)
    - the timestamp of the message that is received in this step (of type `timestamp`)
    - the session IDs of the state of the active participant, where private and public keys are stored (of type `state_id`)
  - Typical *return values* are:
    - the timestamp of the message sent in this step (of type `timestamp`)
    - the session ID of the active participant that has been used or updated in this step (of type `state_id`)
    - The verly last step of the protocol usually doesn't have any return values and hence has return type `unit`.
- The general structure of one protocol step is:

1. Receiving the message:
    1. *receive* the message at the provided timestamp
    ```
    let*? msg = recv_msg msg_ts in
    ```
    2. *decrypt* the message
    ```
    let*? plaintext = pke_dec_with_key_lookup #message_t alice keys_sid key_
    ↪tag cipher in
    ```
    3. *parse* the message to the abstract format
    ```
    let*? abstract_msg = return (parse message_t msg) in
    ```
    4. *check* that the message is of the *expected format* for the step
    ```
    guard_tr (Ping? asbtract_msg);*?
    ```
2. *Find* the *session* of the active participant corresponding to the received message. This includes checks of the received data and the stored data.
    ```
    let*? (st, sid) = lookup_state #state_t alice some_property in
    ```
3. Generate the next message (the reply):
    1. compute the *abstract* reply
    2. *serialize* the reply
    ```
    let wire_msg = serialize #bytes message_t abstract_msg in
    ```
    3. *encrypt* the reply
    ```
    let*? cipher = pke_enc_for alice bob key_sid key_tag plaintext in
    ```
    4. *send* the reply
    ```
    let* reply_ts = send_msg ciper in
    ```
4. *Update* the corresponding *session*
    ```
    set_state alice sid abstract_state;*
    ```
5. *Return* the message timestamp from Step 3.4. and the session ID updated in Step 4.

*Check the model with an example run:*

After implementing the protocol steps, you should check that they model the protocol in the correct way. I.e., call the steps in the right order of an example run of the protocol and closely look at the produced trace:

- In a Run module,
```
module DY.TwoMessage.Run

open DY.Core
open DY.Lib
open DY.Simplified

open DY.TwoMessage.Protocol
```
    - define a run function where you
        * call the protocol steps in the order of an example run of the protocol, passing on arguments (message timestamps, session IDs) as needed.

```
val run:
  unit -> traceful (option unit)
let run () =
  let alice = "Alice" in
  let bob = "Bob" in

  let* (alice_sid, ping_ts) = send_ping alice bob in
  let*? (bob_sid, ack_ts) =
    receive_ping_and_send_ack bob ping_ts in
  receive_ack alice ack_ts;*?
```

* access the trace after the run with `get_trace`

```
let* tr = get_trace in
```

* print the trace with `IO.debug_print_string` using `trace_to_string` with either `default_trace_to_string_printers` or a custom `get_trace_to_string_printers` for pretty printing.

```
let _ = IO.debug_print_string "************* Trace *************\n" in
let _ = IO.debug_print_string (
   trace_to_string default_trace_to_string_printers tr
 ) in
```

- call the `run` function on the empty trace when executing the module

```
let _ = run () empty_trace
```

- Add the `test` target to the `Makefile`.

```
test:
  cd $(TUTORIAL_HOME)/obj; $(FSTAR_EXE) --ocamlenv ocamlbuild -use-ocamlfind -
↪pkg batteries -pkg fstar.lib DY_TwoMessage_Run.native
  $(TUTORIAL_HOME)/obj/DY_TwoMessage_Run.native
```

- Run `make` followed by `make test`.
- Look at the output and check:
  - Did the example run succeed? I.e., do you see any output after the `***Trace***` string? (If not, move `get_trace` and `IO.debug_print_string` around until you find the step that is failing.)
  - Does the trace look like expected? Do all entries appear with the right values?

## 4.5 A Model of the Online? Protocol

Let us now turn to the Online? protocol and implement a protocol model in DY*. The only difference to the Two-Message protocol is that the messages are *encrypted* for the respective participants. So the model of the Online? protocol will be very similar to the previous one of the Two-Message protocol, and we will here focus on the new differences.

First, we discuss how encryption is modelled in DY*.

### 4.5.1 Public Key Encryption in DY*

#### Key storage

We don't model a PKI in DY*, instead we just assume that every participant has some private decryption keys and some public encryption keys associated with other principals in her state. To easily access the keys, we collect all private keys in one session and all public keys in another session. We already saw this, when we first introduced states and in particular *sessions for global information*. So states look like this:

```
State Alice:
  Session 0: collection of private decryption keys
  Session 1: collection of public encryption keys
  ...
```

A participant may have several private keys for different purposes. Think of a protocol with several sub-protocols. For each of the sub-protocols there could be a separate private key which is only used for this sub-protocol. All of those keys are stored in the same session (Session 0 in the example above). To use the correct keys in the protocol steps, we need to store the information which key is used for which purpose. We do this, by tagging the keys. That is, the collection of private keys is a dictionary with the dictionary key being a tag and the values are the keys. So Session 0 in the above example could look like the following (using the notation `lookup_key = value`):

```
Session 0: {
    "SubProt1" = priv_key_for_sub_protocol_1
  , "SubProt2" = priv_key_for_sub_protocol_2
  , "Other" = priv_key_for_some_other_purpose
}
```

Similarly, a participant may have several public encryption keys for different purposes. Additionally, for public keys there are different keys for different other participants. For example, Alice could have two public keys specific to one of the sub-protocols, one for Bob and one for Charlie. Hence, to identify the correct key for encryption, we now use the tag together with the name of the other participant. Thus, the collection of public keys is a dictionary with the dictionary key being the pair of a tag and a participant name. So Session 1 from above could look like the following:

```
Session 1: {
    ("SubProt1", Bob) = pub_key_for_Bob_for_sub_protocol_1
  , ("SubProt1", Charlie) = pub_key_for_Charlie_for_sub_protocol_1
  , ("SubProt2", Bob) = pub_key_for_Bob_for_sub_protocol_2
}
```

In the above examples, we used Session 0 to store Alice's private keys and Session 1 for her public keys. However, the sessions for the keys are not fixed and can be allocated dynamically. Usually this happens in a setup phase before any protocol run.

This setup is *not* part of the protocol *model* itself. Just like protocol runs are not part of the protocol model, but need to be implemented separately (recall *the example run for the Two-Message protocol*). But since the protocol steps in the model need to know where keys are stored for de- and encryption, we need to add these session IDs as additional arguments to the steps. That is, for implementing a protocol step in which you want to encrypt something, you can assume that you get the session ID of the state where public keys are stored as an argument. If you then want to implement an actual run of the protocol, you need to generate those key sessions (and IDs) in a setup phase and pass them on as arguments to the protocol steps in the run. We will see how key setup works later, when we implement an example run for the Online? protocol.

For now, it suffices to remember that private and public keys are stored in separate sessions at a principal and that we use a tag to lookup a private key and a tag participant pair to lookup a public key. With this model of the key storage in mind, we can now look at how encryption and decryption works in DY*.

### Encryption

If Alice wants to encrypt some plaintext for Bob, she needs to lookup Bob's public key in her public keys session, and use this key to encrypt the plaintext.

In `DY.Simplified` there is a function `pke_enc_for` that does those steps. It takes as arguments:

- the participant that does the encryption
- the participant whose public key is used for encryption
- the session ID where the active participant stores her known public keys
- the tag of the key (used to lookup the right key in the public keys session)
- the abstract message

and returns the corresponding ciphertext in wire format. The type is

```
val pke_enc_for:
  #a:Type -> {| parseable_serializeable bytes a |} ->
  (active: principal) -> (for: principal) ->
  (public_keys_sid: state_id) -> (key_tag: string) ->
  (abstract_msg: a) ->
  traceful (option (cipher: bytes))
```

The function fails if there is no key for the respective participant with the given tag stored in the given session of the active participant.

**Example: Encrypting an abstract message**

The `pke_enc_for` function from `DY.Simplified` can be called like this:

```
pke_enc_for #message_t alice bob alices_public_keys_sid key_tag (Ping {alice; n_a})
```

Where Alice wants to encrypt the abstract Ping for Bob using `key_tag` as key to lookup the public key of Bob in her session with the ID `alices_public_keys_sid`, where she stores her known public keys.

### Decryption

Decryption works very similar, where the active participant now has to lookup her private key in her private keys session.

Again, there is a function `pke_dec_with_key_lookup` in `DY.Simplified` for decryption. It takes as arguments:

- the active (decrypting) participant
- the session ID, where the participant stores her private keys
- the tag of the key (used to lookup the right key in the private keys session)
- the ciphertext

and returns the corresponding abstract plaintext. The type is:

```
val pke_dec_with_key_lookup:
  #a:Type -> {| parseable_serializeable bytes a |} ->
  principal ->
  (private_keys_sid: state_id) -> (key_tag: string) ->
```

(continues on next page)

(continued from previous page)

```
  (cipher: bytes) ->
  traceful (option (abstract_plain: a))
```

The function may fail if one of the following is true:

- there is no key with the given tag in the given keys session
- the ciphertext was not encrypted with the key corresponding to the given tag in the given keys session
- parsing of the decrypted plain text fails

**Example: Decrypting a cipher text to an abstract message**

The `pke_dec_with_key_lookup` function from `DY.Simplified` can be called like this:

```
pke_dec_with_key_lookup #message_t bob bobs_private_keys_sid key_tag cipher
```

Here Bob wants to decrypt the `cipher` using the key with tag `key_tag` stored in his session with ID `bobs_private_keys_sid`.

**Remember: Public Key Encryption in DY***

*Key Storage*:

- All private keys are stored in *one* session as a dictionary with dictionary key being some tag.

```
State Alice:
  Session 0: {
      "SubProt1" = priv_key_for_sub_protocol_1
    , "SubProt2" = priv_key_for_sub_protocol_2
    , "Other" = priv_key_for_some_other_purpose
  }
```

- All public keys are stored in *one* session as a dictionary with the dictionary key being a tag together with the name of a participant.

```
State Alice:
  Session 1: {
      ("SubProt1", Bob) = pub_key_for_sub_protocol_1_for_Bob
    , ("SubProt1", Charlie) = pub_key_for_sub_protocol_1_for_Charlie
    , ("SubProt2", Bob) = pub_key_for_sub_protocol_2_for_Bob
  }
```

- The session IDs of the key sessions are generated in a setup phase outside of the protocol steps. Thus, the protocol steps take them as an extra argument.

*Encryption*:

- using `pke_enc_for` from `DY.Simplified`

```
let*? ping_encrypted = pke_enc_for #message_t alice bob alices_public_keys_sid␣
↪key_tag (Ping {alice; n_a}) in
```

- takes the abstract plain text and returns the cipher text in wire format (i.e., includes serializing)

*Decryption*:

- using `pke_dec_with_key_lookup` from `DY.Simplified`

```
let*? ping_plain = pke_dec_with_key_lookup #message_t bob bobs_private_keys_sid␣
↪key_tag cipher in
```

- takes the cipher text in wire format and returns the abstract plain text (i.e., includes parsing)

With this, we can now implement the model of the Online? protocol.

### 4.5.2 Setup

As for the *Two-Message protocol*, create a new folder `Online` and add

- a `Makefile`

```
TUTORIAL_HOME ?= ../..

EXAMPLE_DIRS = path_to/Online

include $(TUTORIAL_HOME)/Makefile
```

- a `DY.Online.Data` module

```
module DY.Online.Data

open Comparse
open DY.Core
open DY.Lib
```

- a `DY.Online.Protocol` module.

```
module DY.Online.Protocol

open Comparse
open DY.Core
open DY.Lib

open DY.Simplified
open DY.Extend

open DY.Online.Data
```

### 4.5.3 The `Data` module

#### The abstract message and state types

We begin the DY* model of the Online? protocol with the abstract message and state types in the `DY.Online.Data` module.

### Messages

**Exercise: Abstract message format**

Define the abstract format for the messages of the Online? protocol as records in DY*.

**Show/Hide Answer**

The messages of the Online? protocol have the same content as the messages in the Two-Message protocol. Hence, we have the same abstract format and can use the same records:

```
type ping_t = {
  alice: principal;
  n_a : bytes;
}

type ack_t = {
  n_a : bytes;
}

type message_t =
  | Ping: (ping:ping_t) -> message_t
  | Ack:  (ack:ack_t) -> message_t
```

**Remember**

The abstract format of messages only defines the *content* of the message.

In particular, the following are *not* part of the abstract format:

- the structure of the message (for example: Is it a pair? Is it a 3-tuple where the second component is a pair?)
- whether the message (or parts of it) are encrypted

**Exercise: Serializing and Parsing for the abstract message format**

Define serializer and parser for the abstract message format using Comparse.

**Show/Hide Answer**

Add `[@@ with_bytes bytes]` to the record types and add

```
%splice [ps_ping_t] (gen_parser (`ping_t))
%splice [ps_ack_t] (gen_parser (`ack_t))
%splice [ps_message_t] (gen_parser (`message_t))

instance parseable_serializeable_bytes_message_t: parseable_serializeable bytes␣
↪message_t =
  mk_parseable_serializeable ps_message_t
```

### States

**Exercise: Abstract state format**

Completely define the abstract state format for states related to the Online? protocol. (You do not need to think about the states where keys are stored.)

**Show/Hide Answer**

Again, the content of the states of the Online? protocol are the same as the ones in the Two-Message protocol:

```
// the record types for the individual states
// and the combined state_t type

[@@ with_bytes bytes]
type sent_ping_t = {
  bob : principal;
  n_a : bytes;
}

[@@ with_bytes bytes]
type sent_ack_t = {
  alice: principal;
  n_a : bytes;
}

[@@ with_bytes bytes]
type received_ack_t = {
  bob : principal;
  n_a : bytes;
}

[@@ with_bytes bytes]
type state_t =
  | SentPing: (ping:sent_ping_t) -> state_t
  | SentAck: (ack:sent_ack_t) -> state_t
  | ReceivedAck: (rack:received_ack_t) -> state_t

// serializing and parsing using Comparse

%splice [ps_sent_ping_t] (gen_parser (`sent_ping_t))
%splice [ps_sent_ack_t] (gen_parser (`sent_ack_t))
%splice [ps_received_ack_t] (gen_parser (`received_ack_t))
%splice [ps_state_t] (gen_parser (`state_t))

instance parseable_serializeable_bytes_state_t: parseable_serializeable bytes state_t␣
↪=
    mk_parseable_serializeable ps_state_t

// the local_state instance

instance local_state_state_t: local_state state_t = {
  tag = "Online.State";
  format = parseable_serializeable_bytes_state_t;
}
```

### Summary

The content of the messages and states of the Online? protocol are the same as for the Two-Message protocol. Hence, the definitions of the abstract formats are the same and your `DY.Online.Data` module should now be the same as the `DY.TwoMessage.Data` module (except for the tag of the states in the `local_state` instance).

### Preparation for Public Key Encryption

Finally, we need a tag for the protocol related keys used for public key en-/decryption of the messages. We define this tag as a global constant also in the `Data` module.

For example:

```
let key_tag = "Online.Key"
```

This concludes the `Data` module and we can continue with the implementation of the protocol steps.

## 4.5.4 The protocol steps

Recall again, that the only difference of the Two-Message and the Online? protocol is that the messages are now encrypted for the respective participants. For this, we use the *encryption function* `pke_enc_for` and the *decryption function* `pke_dec_with_key_lookup`.

### Sending a Ping

As a first attempt, you might try to just copy the `send_ping` step from the model of the Two-Message protocol:

```
1  val send_ping:
2    (alice: principal) -> (bob: principal) ->
3    traceful (state_id & timestamp)
4  let send_ping alice bob =
5    let* n_a = gen_rand in
6
7    let ping = Ping {alice = alice; n_a = n_a} in
8    let ping_wire = serialize message_t ping in
9    let* msg_ts = send_msg ping_wire in
10
11   let ping_state = SentPing {bob = bob; n_a = n_a} in
12   let* sid = start_new_session alice ping_state in
13
14   return (sid, msg_ts)
```

Now that we want to encrypt the message before sending it in Line 9, we need to adapt Line 8. We want to call `pke_enc_for` here instead of the `serialize`.

**Exercise: Preparing the encryption**

What are the arguments to the call of `pke_enc_for` at this point? (Who wants to decrypt for who? Using what key?)

**Show/Hide Answer**

- `alice`
- `bob`
- some state ID of alice, where she stores public encryption keys
- the key tag defined in the `Data` module
- the abstract Ping message `ping`

So the call looks like this:
`pke_enc_for alice bob alice_public_keys_sid key_tag ping`

We observe that the only "undefined" argument is the session ID of the state where Alice stores her public encryption keys. Recall from the *introduction of the key storage model*, that this session ID needs to be added as an extra argument to the protocol step.

**Exercise: Adding the public key session ID as argument**

Add the key session ID where Alice stores her public keys as extra argument to the `send_ping` step.

**Show/Hide Answer**

```
1 val send_ping:
2   (alice: principal) -> (alice_public_keys_sid: state_id) ->
3   (bob: principal) -> traceful (state_id & timestamp)
4 let send_ping alice alice_public_keys_sid bob =
5   ...
```

Now, we want to call `pke_enc_for` in Line 8 and send the resulting cipher text in Line 9. Since encryption is a monadic action, we first need to think about the relevant monads and which `let` to use.

**Exercise: Calling `pke_enc_for` in `send_ping` – the Monads**

1. What monad is the `pke_enc_for` action in?
2. What monad is the `send_ping` function in?
3. What `let` do we have to use in Line 8?

**Show/Hide Answer**

1. `pke_enc_for` is a `traceful` + `option` action (since it may fail, if there is not the right key in the provided session.)
2. `send_ping` is a `traceful` function.
3. None. We can not call `pke_enc_for` inside `send_ping`, since it has more side effects.

Since we want to call `pke_enc_for` in the `send_ping` function, we need to change the monad of `send_ping` to `traceful` + `option`.

**Exercise: Calling pke_enc_for in send_ping**

1. Change the type of `send_ping` so that `send_ping` is a function in the `traceful` + `option` monad.

   **Show/Hide Answer**

```
val send_ping:
  (alice: principal) -> (alice_public_keys_sid: state_id) -> (bob:␣
↪principal) ->
  traceful (option (state_id & timestamp))
```

2. Change Line 8 to use `pke_enc_for`. What `let` do we have to use?

   **Show/Hide Answer**

   Since we want the overall step to fail, if encryption fails, we use `let*?` in Line 8:

```
let*? ping_encrypted = pke_enc_for alice bob alice_public_keys_sid key_tag ping␣
↪in
```

3. Is there anything else we need to change in the `send_ping` function?

   **Show/Hide Answer**

   As we changed the monad of `send_ping` to `traceful` + `option`, we now need to return an `option`:

```
return (Some (sid, msg_ts))
```

Putting everything together, the `send_ping` step for the Online? protocol looks like this (highlighting the changes to the `send_ping` step from the Two-Message protocol):

```
1  val send_ping:
2    (alice: principal) -> (alice_public_keys_sid: state_id) -> (bob: principal) ->
3    traceful (option (state_id & timestamp))
4  let send_ping alice alice_public_keys_sid bob =
5    let* n_a = gen_rand in
6
7    let ping = Ping {alice = alice; n_a = n_a} in
8    let*? ping_encrypted = pke_enc_for alice bob alice_public_keys_sid key_tag ping in
9    let* msg_ts = send_msg ping_encrypted in
10
11   let ping_state = SentPing {bob = bob; n_a = n_a} in
12   let* sid = start_new_session alice ping_state in
13
14   return (Some (sid, msg_ts))
```

**Common Mistakes**

If you changed Line 8 to

```
let*? ping_encrypted = pke_enc_for alice bob alice_public_keys_sid key_tag ping in
```

and get the error

```
- Expected type traceful (option (*?u8*) _)
  but
      ...
  has type traceful (state_id & timestamp)
```

you probably forgot to

- add `option` to the monad of `send_ping`:

```
val send_ping:
  (alice: principal) -> (alice_public_keys_sid: state_id) -> (bob: principal) ->
  traceful (option (state_id & timestamp))
```

- or to return an `option`:

```
return (Some (sid, msg_ts))
```

### Replying with an Ack

As for `send_ping`, we will build the second protocol step for the Online? protocol from the second step of the Two-Message protocol and focus on the differences.

Start by copying the `receive_ping_and_send_ack` step from `DY.TwoMessage.Protocol`:

```
1  val receive_ping_and_send_ack:
2     principal -> timestamp ->
3     traceful (option (state_id & timestamp))
4  let receive_ping_and_send_ack bob msg_ts =
5    let*? msg = recv_msg msg_ts in
6    let*? png_ = return (parse message_t msg) in
7    guard_tr (Ping? png_);*?
8
9    let Ping png = png_ in
10   let alice = png.alice in
11   let n_a = png.n_a in
12
13   let ack = Ack {n_a} in
14   let* ack_ts = send_msg (serialize message_t ack) in
15
16   let ack_state = SentAck {alice; n_a} in
17   let* sess_id = start_new_session bob ack_state in
18
19   return (Some (sess_id, ack_ts))
```

The main differences between the Online? and the Two-Message protocol are:

- the received Ping is encrypted and hence Bob needs to decrypt the message in Line 6
- the Ack needs to be encrypted for Alice in Line 14

### Decrypting the Ping

We begin with decrypting the received message. For this we want to use the *decryption function* `pke_dec_with_key_lookup`.

**Exercise: Preparing the Decryption**

What are the arguments to the call of `pke_dec_with_key_lookup` in Line 6? (Who wants to decrypt? Using what key?)

**Show/Hide Answer**

- `bob`
- some state ID of Bob, where he stores his private decryption keys
- the key tag defined in the `Data` module
- the wire-format ciphertext `msg`

So the call looks like this:

```
pke_dec_with_key_lookup #message_t bob bob_private_keys_sid key_tag msg
```

Again, we currently don't have Bob's private key session ID available and need to add it as extra argument to the protocol step. So the type of the step changes to:

```
val receive_ping_and_send_ack:
  principal -> state_id -> timestamp ->
  traceful (option (state_id & timestamp))
let receive_ping_and_send_ack bob bob_private_keys_sid msg_ts =
  ...
```

We are now ready to call the decryption function.

**Exercise: Calling `pke_dec_with_key_lookup` in `receive_ping_and_send_ack`**

1. Can we call `pke_dec_with_key_lookup` inside `receive_ping_and_send_ack`? (Think about the relevant monads.)

   **Show/Hide Answer**

   Yes. Both the decryption action and the overall function are in the `traceful` + `option` monad.

2. What `let` do we need to use in Line 6?

   **Show/Hide Answer**

   Since we want the protocol step to fail, if decryption fails, we need to use `let*?`.

3. Change Line 6 accordingly.

   **Show/Hide Answer**

   ```
   let*? png_ = pke_dec_with_key_lookup #message_t bob private_keys_sid key_tag msg␣
   ↪in
   ```

   Recall, that decryption includes parsing, so the returned `png_` is already in the abstract message format and we don't need to explicitly call `parse`.

This concludes decryption of the received message to an abstract Ping message.

### Encrypting the Ack

Encrypting the Ack message works just like encrypting the Ping in the first protocol step.

**Exercise: Encrypting the Ack**

Change the step so that the Ack is encrypted for Alice.

You may want to think about the following and adapt the model accordingly:

1. What are the arguments for the encryption function?

   **Show/Hide Answer**

   - `bob`
   - `alice`
   - a session ID where Bob stores his public encryption keys
   - the key tag defined in the `Data` module
   - the abstract Ack `ack`

2. Does the type of the protocol step need to change?

   **Show/Hide Answer**

   Yes, we need to add the session ID for the public keys of Bob as extra argument:

   ```
   val receive_ping_and_send_ack:
     principal -> state_id -> state_id ->
     timestamp -> traceful (option (state_id & timestamp))
   let receive_ping_and_send_ack bob bob_private_keys_sid bob_public_keys_sid msg_
   ↪ts =
     ...
   ```

3. Can we call the encryption function here and what `let` should we use?

   **Show/Hide Answer**

   Yes, since both the encryption action and the overall protocol step are in the `traceful` + `option` monad. Since we want the whole step to fail, if encryption fails, we use `let*?`.

   Line 14 changes to:

   ```
   let*? ack_encrypted = pke_enc_for bob alice bob_public_keys_sid key_tag ack in
   ```

### Summary

To summarize, we

- used `pke_dec_with_key_lookup` for decrypting the received message (adding Bob's private keys session ID as extra argument to the step)
- used `pke_enc_for` to encrypt the Ack for Alice (adding Bob's public keys session ID as extra argument)

so that the final second protocol step now looks like this (highlighting the changes to the second step of the Two-Message protocol):

```
val receive_ping_and_send_ack:
  principal -> state_id -> state_id -> timestamp ->
  traceful (option (state_id & timestamp))
let receive_ping_and_send_ack bob bob_private_keys_sid bob_public_keys_sid msg_ts =
  let*? msg = recv_msg msg_ts in
  let*? png_ = pke_dec_with_key_lookup #message_t bob bob_private_keys_sid key_tag msg in
  guard_tr (Ping? png_);*?

  let Ping png = png_ in
  let alice = png.alice in
  let n_a = png.n_a in

  let ack = Ack {n_a} in
  let*? ack_encrypted = pke_enc_for bob alice bob_public_keys_sid key_tag ack in
  let* ack_ts = send_msg ack_encrypted in

  let ack_state = SentAck {alice; n_a} in
  let* sess_id = start_new_session bob ack_state in

  return (Some (sess_id, ack_ts))
```

### Receiving an Ack

The final step of the Online? protocol is again almost the same as the final step of the Two-Message protocol. The only difference is, that the received Ack is encrypted and needs to be decrypted by Alice. This works in the same way as in the second step, where Bob decrypts the Ping. Looking back at the previous step, you should be able to adapt the final step of the Two-Message protocol.

**Exercise: Adapting `receive_ack` from the model of the Two-Message protocol**

Implement the final protocol step by adapting the `receive_ack` step from the model of the Two-Message protocol.

You can follow these steps:

1. Copy the `receive_ack` step from the model of the Two-Message protocol. Where do we need to perform decryption?

   **Show/Hide Answer**

```
val receive_ack:
  principal -> timestamp ->
  traceful (option state_id)
let receive_ack alice ack_ts =
  let*? msg = recv_msg ack_ts in
```

```
6  let*? ack = return (parse message_t msg) in
7  guard_tr (Ack? ack);*?
8
9  let Ack ack = ack in
10 let n_a = ack.n_a in
11
12 let*? (sid, st) = lookup_state #state_t alice
13   (fun st ->
14         SentPing? st
15     && (SentPing?.ping st).n_a = n_a
16   ) in
17 guard_tr(SentPing? st);*?
18 let bob = (SentPing?.ping st).bob in
19
20 set_state alice sid (ReceivedAck {bob; n_a});*
21
22 return (Some sid)
```

2. What are the arguments of the decryption function?

   **Show/Hide Answer**

   - `alice`
   - the session ID where Alice stores her private decryption keys
   - the key tag defined in the `Data` module
   - the wire-format ciphertext `msg`

3. Does the type of the protocol step need to change?

   **Show/Hide Answer**

   Yes. We need to add the private keys session ID as extra argument:

   ```
   val receive_ack:
     principal -> state_id -> timestamp ->
     traceful (option state_id)
   let receive_ack alice alice_private_keys_sid ack_ts =
     ...
   ```

4. Can we call the decryption function here and what `let` should we use?

   **Show/Hide Answer**

   Yes. Since both decryption and the step are in the `traceful` + `option` monad. We use `let*?`, as we want the whole step to fail if decryption fails.

   Line 6 changes to:

   ```
   let*? ack = pke_dec_with_key_lookup #message_t alice alice_private_keys_sid key_
   ↪tag msg in
   ```

   We do not need to call parsing explicitly, as this is already part of decryption.

The final step of the Online? protocol looks like this (highlighting the changes to `receive_ack` from the Two-Message protocol):

```
val receive_ack:
  principal -> state_id -> timestamp ->
  traceful (option state_id)
let receive_ack alice alice_private_keys_sid ack_ts =
  let*? msg = recv_msg ack_ts in
  let*? ack = pke_dec_with_key_lookup #message_t alice alice_private_keys_sid key_tag␣
↪msg in
  guard_tr (Ack? ack);*?

  let Ack ack = ack in
  let n_a = ack.n_a in

  let*? (sid, st) = lookup_state #state_t alice
    (fun st ->
          SentPing? st
      && (SentPing?.ping st).n_a = n_a
    ) in
  guard_tr(SentPing? st);*?
  let bob = (SentPing?.ping st).bob in

  set_state alice sid (ReceivedAck {bob; n_a});*

  return (Some sid)
```

This concludes our model of the Online? protocol. Which is very similar to the model of the Two-Message protocol, with the new features of de- and encryption of messages.

### 4.5.5 An example protocol run

To check that our model works as expected, we implement an example run, where Alice sends a Ping to Bob, Bob replies with an Ack and Alice receives the Ack and checks that she started a run with the received nonce for Bob.

The general structure is the same as for the *Two-Message protocol*: we add a `Run` module in which we define a `run` function where we call the protocol steps in the right order passing on arguments as needed and finally print the trace.

But since we now use de- and encryption in the protocol steps, we also need to take care of the key storage setup, which will be our main focus in this section.

But first, let's setup the example run module.

#### The `Run` module

**Exercise: Setting up the `Run` module**

Follow the steps from *How-To: Modelling a Protocol* to add a new `Run` module, add the `test` target to the `Makefile`, implement an (empty) `run` function that is called when executing the module and in which the current trace is printed.

Try your setup by running `make` followed by `make test`.

**Show/Hide Answer**

The `Run` module:

```
module DY.Online.Run

open DY.Core
open DY.Lib
open DY.Simplified

open DY.Online.Data
open DY.Online.Protocol

let run () : traceful (option unit ) =
  let _ = IO.debug_print_string "************* Trace *************\n" in

  let* tr = get_trace in
  let _ = IO.debug_print_string (
      trace_to_string default_trace_to_string_printers tr
    ) in

  return (Some ())

let _ = run () empty_trace
```

Add to the `Makefile`:

```
test:
	cd $(TUTORIAL_HOME)/obj; $(FSTAR_EXE) --ocamlenv ocamlbuild -use-ocamlfind -
↪pkg batteries -pkg fstar.lib DY_Online_Run.native
	$(TUTORIAL_HOME)/obj/DY_Online_Run.native
```

Running `make` followed by `make test` produces several lines on the terminal where the final line is:

```
************* Trace *************
```

Now that we have a working `Run` module, we will populate the `run` function. In the end, we want to call the protocol steps, so we have to define all necessary arguments for those calls. These arguments are

- the participants
- the key session IDs

For the participants, we define constants

```
let alice = "Alice" in
let bob = "Bob" in
```

just as in the example run of the Two-Message protocol.

Let's now take a closer look at how to setup the key storage for de- and encryption.

### Setup of Key Storage

Recall the layout of key storage at the participants from *before*. For our example run of the Online? protocol, we want to have the following states after key storage setup:

```
State Alice:
  Session alice_private_keys_sid: {
    "Online.Key" = alice_private_key
  }
  Session alice_public_keys_sid: {
    ("Online.Key", Bob) = bob_public_key
  }
```

Alice has one private key and one public key for Bob for the Online? protocol in her state. And similarly for Bob:

```
State Bob:
  Session bob_private_keys_sid: {
    "Online.Key" = bob_private_key
  }
  Session bob_public_keys_sid: {
    ("Online.Key", Alice) = alice_public_key
  }
```

To achieve these state layouts, the setup consists of three phases for every participant:

1. Allocate sessions for the private and public keys.
2. Generate and store own private keys.
3. Retrieve and store public keys from other participants.

#### 1. Allocating sessions for private and public keys

In the first initialization step, we allocate new sessions at the participants for the private and public keys. These new sessions are initialized as empty dictionaries.

In `DY.Lib` there are `initialize_*` functions that take as argument a participant and return the new session ID where the keys are to be stored. They are `traceful` functions that can not fail, since they just add new sessions to the state.

**Example: Allocate key sessions for Alice**

For the private keys session of Alice we use `initialize_private_keys`:

```
let* alice_private_keys_sid = initialize_private_keys alice in
```

and for the public keys session we use `initialize_pki`:

```
let* alice_public_keys_sid = initialize_pki alice in
```

After these initialization steps the state of Alice looks like this:

```
State Alice:
  Session alice_private_keys_sid: { }
  Session alice_public_keys_sid: { }
```

Alice has two new sessions, each containing an empty dictionary.

**Exercise: Allocate key sessions for Bob**

Allocate new sessions for the private and public keys for Bob.

**Show/Hide Answer**

```
let* bob_private_keys_sid = initialize_private_keys bob in
let* bob_public_keys_sid = initialize_pki bob in
```

### 2. Generating and Storing private keys

Now that we know, *where* to store the private keys, we can actually generate and store them.

In `DY.Simplified` there is a corresponding function:

```
val generate_private_pke_key:
  (alice: principal) -> (alice_private_keys_sid: state_id) -> (key_tag: string) ->
  traceful (option unit)
```

The function takes as arguments

- the participant to generate the private key for
- the session ID where the key should be stored, and
- the tag under which the key should be stored.

It generates a new key and stores it under the given tag in the provided session ID of the participant. The step may fail, if the session at the provided ID is not a private key dictionary.

TODO: Explain why we use ;* and not ;*? here: It can fail but we don't do anything different if it does.

**Example: Generate private key for Alice**

To generate the private key for the Online? protocol for Alice, we use the session ID that we just created in step 1. and the key tag specified in the `Data` module:

```
generate_private_pke_key alice alice_private_keys_sid key_tag;*
```

After this step, the state of Alice looks like this:

```
State Alice:
  Session alice_private_keys_sid: {
    "Online.Key" = alice_private_key
  }
  Session alice_public_keys_sid: { }
```

**Exercise: Generate private key for Bob**

Generate a private key for the Online? protocol for Bob.

**Show/Hide Answer**

```
generate_private_pke_key bob bob_private_keys_sid key_tag;*
```

### 3. Retrieving and Storing public keys

Now that both participants have a private key, we want to store the corresponding public keys at the other participant.

Again, there is a corresponding function in `DY.Simplified`:

```
val install_public_pke_key:
  (alice: principal) -> (alice_public_keys_sid: state_id) ->
  (key_tag: string) ->
  (bob: principal) -> (bob_private_keys_sid: state_id) ->
  traceful (option unit)
```

Let's slowly go through the arguments of this function:

- `alice` is the participant who wants to store the public key
- `alice_public_keys_sid` is the session ID (of the public keys session), where Alice wants to store the key
- `key_tag` is the tag of the key used
  - as part of the lookup key in Alice's public keys session for the new key
  - to lookup the private key at Bob
- `bob` is the participant whose public key should be stored
- `bob_private_keys_sid` is the session ID of the private keys session of Bob, where he stores his private keys

A call to the function

```
install_public_pke_key alice alice_public_keys_sid key_tag bob bob_private_keys_sid
```

should be read as:

> `alice` wants to store in her public keys session (with ID `alice_public_keys_sid`) under the lookup key pair `(key_tag, bob)` the public key related to the private key that `bob` stores in his private keys session `bob_private_keys_sid` under the same tag `key_tag`.

**Example: Storing Bob's public key at Alice**

Assume that after generation of the private keys, the states of Alice and Bob look like the following:

```
State Alice:
  Session alice_private_keys_sid: {
    "Online.Key" = alice_private_key
  }
  Session alice_public_keys_sid: { }

State Bob:
  Session bob_private_keys_sid: {
      "Online.Key" = bob_private_key
    , "Other.Key" = bob_other_private_key
  }
  Session bob_public_keys_sid: { }
```

Alice has one private key for the Online? protocol, and Bob has two private keys: one for the Online? protocol and another one for some other protocol.

If Alice wants to store the public key of Bob for the Online? protocol, she calls the `install_public_pke_key` function with the following arguments:

```
install_public_pke_key alice alice_public_keys_sid "Online.Key" bob bob_private_keys_
↪sid
```

After this call, Alice has one entry in her public keys session:

```
State Alice:
  Session alice_public_keys_sid: {
    ("Online.Key", Bob) = public key corresponding to bob_private_key
  }
```

You may ask yourself at this point, what the value of the public key actually is and where it comes from. After all, so far we only generated *private* keys. At this point, it is enough to know that the public key can be computed from the corresponding private key. We don't need to care about how exactly that works. Just remember: A public key is identified by its corresponding private key.

**I want to know how it works.**

Since we are in the *symbolic* world, the public keys are actually just defined by adding a "public key of" constructor to the private key. That is, the public key corresponding to the private key `private_key` is `Pk private_key`.

**Exercise: Storing Alice's public key at Bob**

Store the public key of Alice at Bob.

**Show/Hide Answer**

```
install_public_pke_key bob bob_public_keys_sid key_tag alice alice_private_keys_sid;*
```

Note that here we use the constant `key_tag` defined in the `Data` module, in contrast to the explicit string `"Online.Key"` in the above example for Alice. Both is correct, but it is usually better to use the constants.

### Summary of Key Storage Setup

This concludes setup of the key storage for our example run of the Online? protocol. Your `run` function should look like the following:

```
let run () : traceful (option unit) =
  let _ = IO.debug_print_string "************* Trace *************\n" in

  let alice = "Alice" in
  let bob = "Bob" in

  (*** key storage setup ***)

  let* alice_private_keys_sid = initialize_private_keys alice in
  let* alice_public_keys_sid = initialize_pki alice in

  let* bob_private_keys_sid = initialize_private_keys bob in
  let* bob_public_keys_sid = initialize_pki bob in

  generate_private_pke_key alice alice_private_keys_sid key_tag;*
  generate_private_pke_key bob bob_private_keys_sid key_tag;*
```

(continues on next page)

(continued from previous page)

```
  install_public_pke_key alice alice_public_keys_sid key_tag bob bob_private_keys_sid;*
  install_public_pke_key bob bob_public_keys_sid key_tag alice alice_private_keys_sid;*

  let* tr = get_trace in
  let _ = IO.debug_print_string (
      trace_to_string default_trace_to_string_printers tr
    ) in

  return (Some ())
```

If you run `make` followed by `make test` now, you should see the following trace at the end of the output:

```
************* Trace *************
{"TraceID": 0, "Type": "Session", "SessionID": 0, "Principal": "Alice", "Tag": "DY.Lib.
↪State.PrivateKeys", "Content": "[]"}
{"TraceID": 1, "Type": "Session", "SessionID": 1, "Principal": "Alice", "Tag": "DY.Lib.
↪State.PKI", "Content": "[]"}
{"TraceID": 2, "Type": "Session", "SessionID": 0, "Principal": "Bob", "Tag": "DY.Lib.
↪State.PrivateKeys", "Content": "[]"}
{"TraceID": 3, "Type": "Session", "SessionID": 1, "Principal": "Bob", "Tag": "DY.Lib.
↪State.PKI", "Content": "[]"}
{"TraceID": 4, "Type": "Nonce", "Usage": {"Type": "PkeKey", "Tag": "Online.Key", "Data":
↪"[Alice,]"}}
{"TraceID": 5, "Type": "Session", "SessionID": 0, "Principal": "Alice", "Tag": "DY.Lib.
↪State.PrivateKeys", "Content": "[Online.Key = (Nonce #4),]"}
{"TraceID": 6, "Type": "Nonce", "Usage": {"Type": "PkeKey", "Tag": "Online.Key", "Data":
↪"[Bob,]"}}
{"TraceID": 7, "Type": "Session", "SessionID": 0, "Principal": "Bob", "Tag": "DY.Lib.
↪State.PrivateKeys", "Content": "[Online.Key = (Nonce #6),]"}
{"TraceID": 8, "Type": "Session", "SessionID": 1, "Principal": "Alice", "Tag": "DY.Lib.
↪State.PKI", "Content": "[(Online.Key, Bob) = (Pk(sk=(Nonce #6))),]"}
{"TraceID": 9, "Type": "Session", "SessionID": 1, "Principal": "Bob", "Tag": "DY.Lib.
↪State.PKI", "Content": "[(Online.Key, Alice) = (Pk(sk=(Nonce #4))),]"}
```

**Hints for reading the trace output**

- Nonces are referred to by their creation time. As such we see `Nonce #ts`, where `ts` is the timestamp of creation.
- Dictionaries are represented as lists with entries `key = value`.
- Notation for a public key corresponding to a secret key is `Pk (sk = the_private_key)`.

Let's look at the entries one by one:

- *Trace entries 0 - 3* correspond to initialization of the key sessions for Alice and Bob. Both get two empty dictionaries in Sessions 0 and 1 for private and public keys respectively.
- *Entries 4 + 5* show the generation (4) and storing (5) of Alice's private key.
  - Entry 4 reads: Created a new nonce that is to be used as private key with the tag "Online.Key" for Alice.
  - Entry 5 shows Session 0 of Alice, where the new key (the nonce generated in Step 4) is stored under the

tag "Online.Key"

- *Entries 6 + 7* show the generation (6) and storing (7) of Bob's private key.
- *Entry 8* shows storing of Bob's public key in Alice's public key session (Session 1). The dictionary now contains one entry with the lookup key "Online.Key" and "Bob" and the value is the public key corresponding to the nonce generated in Step 6, i.e., the private key of Bob.
- *Entry 9* shows storing of Alice's public key in Bob's public key session.

### How-To: Setup Key Storage

We summarize the key setup for later reference:

**How-To: Setup Key Storage**

1. Allocate sessions for the private and public keys:

```
let* alice_private_keys_sid = initialize_private_keys alice in
let* alice_public_keys_sid = initialize_pki alice in
```

2. Generate and store own *private* keys:

```
generate_private_pke_key alice alice_private_keys_sid key_tag;*
```

3. Retrieve and store *public* keys from other participants:

```
install_public_pke_key alice alice_public_keys_sid key_tag bob bob_private_keys_
↪sid;*
```

(Public keys are identified by their corresponding private key.)

### The actual protocol run

We now have all things set up to call the protocol steps in the `run` function.

**Exercise: Call the protocol steps**

In the `run` function, call the protocol steps corresponding to a run where Alice sends a Ping to Bob, Bob replies with an Ack, and Alice receives the Ack.

Carefully think about the needed arguments, in particular the keys session IDs.

**Show/Hide Answer**

```
let*? (alice_sid, ping_ts) = send_ping alice bob alice_public_keys_sid in
let*? (bob_sid, ack_ts) = receive_ping_and_send_ack bob bob_private_keys_sid bob_
↪public_keys_sid ping_ts in
receive_ack alice alice_private_keys_sid ack_ts;*?
```

Before calling `make test`, we again use a pretty printing function for the protocol messages and states:

- Download the `DY.Online.Run.Printing.fst` file from the Tutorial Repo on GitHub,
- import the module in your `Run` module and
- change the default printers to `get_trace_to_string_printers`

so that printing is now:

```
let _ = IO.debug_print_string (
    trace_to_string get_trace_to_string_printers tr
 ) in
```

If you run `make` followed by `make test` now, the output should end with the following trace which is quite long and we will go through it line by line:

```
************* Trace *************
{"TraceID": 0, "Type": "Session", "SessionID": 0, "Principal": "Alice", "Tag": "DY.Lib.
↪State.PrivateKeys", "Content": "[]"}
{"TraceID": 1, "Type": "Session", "SessionID": 1, "Principal": "Alice", "Tag": "DY.Lib.
↪State.PKI", "Content": "[]"}
{"TraceID": 2, "Type": "Session", "SessionID": 0, "Principal": "Bob", "Tag": "DY.Lib.
↪State.PrivateKeys", "Content": "[]"}
{"TraceID": 3, "Type": "Session", "SessionID": 1, "Principal": "Bob", "Tag": "DY.Lib.
↪State.PKI", "Content": "[]"}
{"TraceID": 4, "Type": "Nonce", "Usage": {"Type": "PkeKey", "Tag": "Online.Key", "Data":
↪"[Alice,]"}}
{"TraceID": 5, "Type": "Session", "SessionID": 0, "Principal": "Alice", "Tag": "DY.Lib.
↪State.PrivateKeys", "Content": "[Online.Key = (Nonce #4),]"}
{"TraceID": 6, "Type": "Nonce", "Usage": {"Type": "PkeKey", "Tag": "Online.Key", "Data":
↪"[Bob,]"}}
{"TraceID": 7, "Type": "Session", "SessionID": 0, "Principal": "Bob", "Tag": "DY.Lib.
↪State.PrivateKeys", "Content": "[Online.Key = (Nonce #6),]"}
{"TraceID": 8, "Type": "Session", "SessionID": 1, "Principal": "Alice", "Tag": "DY.Lib.
↪State.PKI", "Content": "[(Online.Key, Bob) = (Pk(sk=(Nonce #6))),]"}
{"TraceID": 9, "Type": "Session", "SessionID": 1, "Principal": "Bob", "Tag": "DY.Lib.
↪State.PKI", "Content": "[(Online.Key, Alice) = (Pk(sk=(Nonce #4))),]"}
{"TraceID": 10, "Type": "Nonce", "Usage": {"Type": "NoUsage"}}
{"TraceID": 11, "Type": "Nonce", "Usage": {"Type": "PkeNonce"}}
{"TraceID": 12, "Type": "Message", "Content": "PkeEnc(pk=(Pk(sk=(Nonce #6))),␣
↪nonce=(Nonce #11), msg=([Alice, [Nonce #10,]]))"}
{"TraceID": 13, "Type": "Session", "SessionID": 2, "Principal": "Alice", "Tag": "Online.
↪State", "Content": "SentPing [n_a = (Nonce #10), to = (Bob)]"}
{"TraceID": 14, "Type": "Nonce", "Usage": {"Type": "PkeNonce"}}
{"TraceID": 15, "Type": "Message", "Content": "PkeEnc(pk=(Pk(sk=(Nonce #4))),␣
↪nonce=(Nonce #14), msg=( [Nonce #10,]))"}
{"TraceID": 16, "Type": "Session", "SessionID": 2, "Principal": "Bob", "Tag": "Online.
↪State", "Content": "SentAck [n_a = (Nonce #10), to = (Alice)]"}
{"TraceID": 17, "Type": "Session", "SessionID": 2, "Principal": "Alice", "Tag": "Online.
↪State", "Content": "ReceivedAck [n_a = (Nonce #10), from = (Bob)]"}
```

**Common Mistakes**

If you don't see a trace in your output, one of the steps fails. Move around the trace printing part (including `get_trace`) and check which of the steps it is (the first one, for which you don't see a trace after `make` + `make test`). Once you found the step, ask yourself:

- Did you use the right arguments for that step?
- Did you maybe swap private and public keys session IDs?

**A surprising Non-Mistake**

The run in the current `run` function will be successful (i.e., you'll see the correct trace) if you swapped Alice's keys session IDs for Bob's. For example, in the call to `receive_ack` you can use `bob_private_keys_sid` instead of `alice_private_keys_sid`.

Why is that possible? In the `receive_ack` step, Alice needs to decrypt the Ack and hence is looking for a key with the "Online.Key" tag in the provided session, which is now seemingly Bob's private key session. So apparently, Alice can decrypt a message encrypted for her using Bob's private key. Is there a bug in DY*'s decryption?

**Show/Hide Answer**

Of course there is no bug in DY*'s decryption!

The misleading part in the above argument is that we don't provide sessions but session **IDs**. So Alice does not look for the key in Bob's private keys session, instead she looks in her own state at the provided session **ID**. Since both `alice_private_keys_sid` and `bob_private_keys_sid` are 0 in our run, she looks for the private key in her own Session 0 in both cases.

*Lesson learned:* Session IDs are just numbers and a participant can only look at sessions in his own state. The same session ID may appear in the states of different participants, but probably with very different content related to it.

Let's now look at the trace entries:

- *Entries 0-9* show the key storage setup that we previously saw.
- *Entry 10 - 13* come from the first step of the protocol, where Alice generates a nonce $n_A$ and sends this nonce together with her name to Bob.
  - *Entry 10* shows the generation of the nonce $n_A$ TODO: what to say about the usage/type? Ideally, I want to avoid talking about this. But it is needed for private key generation (and maybe PkeNonces)
  - Let's now first look at *Entry 12*:

    ```
    {"TraceID": 12, "Type": "Message", "Content": "PkeEnc(pk=(Pk(sk=(Nonce #6))),␣
    ↪nonce=(Nonce #11), msg=([Alice, [Nonce #10,]]))"}
    ```

    This is the encrypted message from Alice to Bob.

    * At the very end we see the plaintext `msg=([Alice, [Nonce #10,]])` which is the message containing Alice's name and the nonce $n_A$ she generated at time 10.
    * In the beginning of the message content we see `PkeEnc` which means that the plaintext is encrypted.
    * The public encryption key that is used for this message, is given as we have seen previously: `pk=(Pk(sk=(Nonce #6)))`. It is the public key corresponding to the private key that was generated at time 6. If we look at the entry at time 6, we see that this is (as expected) the private key of Bob.
    * We are left with the middle part `nonce=(Nonce #11)`. TODO: ideally i don't want to talk about this nonce (and hence entry 11) at all
  - *Entry 13* shows that Alice creates a new session with ID 2 for the protocol status, where she stores that she sent the Ping $n_A$ (`Nonce #10`) to Bob.
- *Entries 14 - 16* come from the second protocol step, where Bob receives the Ping, and replies with an Ack.
  - *Entry 15* shows the encrypted Ack

    ```
    {"TraceID": 15, "Type": "Message", "Content": "PkeEnc(pk=(Pk(sk=(Nonce #4))),␣
    ↪nonce=(Nonce #14), msg=( [Nonce #10,]))"}
    ```

    with

    - the plaintext `msg=( [Nonce #10,])` being the nonce generated at time 10 (the nonce $n_A$, Alice generated and sent to Bob in the Ping)
    - the encryption key `pk=(Pk(sk=(Nonce #4)))` being the public key related to the private key generated at time 4 (which is, as expected, the private key of Alice)
    - the encryption nonce `nonce=(Nonce #14)` generated in the previous step (time 14)
  - *Entry 16* shows the new session that Bob creates in the step, where he stores that he sent the Ack $n_A$ (`Nonce #10`) to Alice.
- *Entry 17* shows the updated Session 2 of Alice, where she now stores that she received the Ack with $n_A$ (`Nonce #10`) from Bob.

So we see that our model of the Online? protocol seems to be correct. We can successfully model an example run, where everything works as expected. This concludes our model of the Online? protocol.

## 4.6 A First Implementation of NSL

Looking back at the model of the Online? protocol, you should now be able to implement a first model of the *NSL protocol*.

**Show/Hide Answer**

https://github.com/REPROSEC/dolev-yao-star-tutorial-code/tree/main/examples/NSL

CHAPTER

FIVE

# STATING SECURITY PROPERTIES

In this chapter, we learn how to state security properties in DY* and what they mean.

In DY*, properties are expressed as properties on all traces complying with the protocol of interest. As an example, recall the intuitive description of the secrecy property for the Online? protocol from the *introduction*:

> A nonce $n_A$ that Alice generates for (and sends to) some honest other party Bob, is only known to Alice and Bob.

In the trace-based view this is expressed as:

> For every trace complying with the Online? protocol, the attacker does not get to know the nonce $n_A$ that Alice generated for Bob, as long as both Alice and Bob are honest.

This raises the immediate questions:

1. How does the attacker get to know values?
2. What does it mean for participants to be honest?
3. How can we express that a trace complies with the protocol?

We answer the first two questions in this chapter, by describing DY*'s *attacker model*. The detailed answer to the last question is deferred to the next chapter.

After these preliminaries, we show how to formally state security properties in DY* using the most common properties of *secrecy* and *authentication* as examples.

## 5.1 Attacker Knowledge and Corruption

### 5.1.1 Attacker Knowledge

In DY* we consider an active network attacker that can read messages from the trace, corrupt states and derive new terms from these. With "attacker knowledge", we denote the set of all terms that the attacker can construct. The attacker knowledge is defined recursively in the `attacker_knows_aux` function in `DY.Core.Attacker.Knowledge` which we will now discuss on a high level.

The base case consists of three possibilities:

- The attacker knows any *message* that is sent over the network, i.e., the content of every message entry on the trace.
- The attacker knows any *constant*, i.e., values like strings and numbers. (That is, he can construct any string.)
- The attacker knows the content of any *state*, that he corrupted (via a corruption entry on the trace).

From this immediate knowledge he can read from the trace, and any prior knowledge he has, he can construct more terms by applying cryptographical functions. TODO: ignoring concat and split on purpose! since we didn't talk about our symbolic terms so far, and i don't know where we actually need this For example for public key encryption:

- If the attacker knows a private key, he also knows the corresponding public key.
- If the attacker knows the public key, the nonce and the plaintext, he can construct the ciphertext corresponding to the public key encryption of the plaintext using the public key and the nonce.
- If the attacker knows the private key and a ciphertext, he can construct the decrypted plaintext corresponding to the ciphertext under that private key.

And there are similar intuitive derivation rules for symmetric encryption (AEAD), signatures, hashes, Diffie-Hellman, KDFs and KEM, which we will not go into details here.

**Remember: Attacker Knowledge**

The attacker knows any

- message on the trace,
- constant and
- state, he corrupted.

From this knowledge, he can build more terms by applying cryptographic functions. For example public key de-/encryption and related keys.

### 5.1.2 Corruption

In DY* we have a fine-grained corruption model, where the attacker can corrupt single states of a participant, a whole session of a participant, or the full state of a participant.

TODO: Work in progress: continue from here

Now that we have a good intuition for security properties and the attacker capabilities, we turn to formally stating the two most common security properties in DY*: Secrecy and Authentication.

## 5.2 Secrecy

Secrecy is a common and important property, saying that some value stays secret during runs of a protocol, as long as all participants are honest.

For example, for the Online? protocol, we want to show secrecy of the nonce $n_A$ that is generated by Alice. In the beginning of this chapter, we had the intuitive trace-based description of secrecy:

> For every trace complying with the Online? protocol, the attacker does not get to know the nonce $n_A$ that Alice generated for Bob, as long as both Alice and Bob are honest.

If we spell out the quantifiers and logical connectives explicitly in the above description, we can state the property as:

> $\forall$ trace $tr$ $\forall$ Alice, Bob, nonce $n_A$:
>
> ($tr$ complies with the Online? protocol $\wedge$ Alice generated $n_A$ for Bob on $tr$) $\implies$
>
> (Alice and Bob are honest on $tr$ $\implies$ attacker does not know $n_A$ from $tr$).

We have seen previously in the attacker model, that we describe things the attacker knows and how he corrupts participants. This is the contrary to "honest" (¬ is corrupt) and "attacker does not know" (¬ (attacker knows)). We hence rephrase the last line to the equivalent:

attacker knows $n_A$ from $tr$ $\implies$ one of Alice or Bob is corrupt on $tr$

Now recall that we can express a formula of the form

$\forall$ variables: pre_condition variables $\implies$ post_condition variables

as an F* `Lemma`:

```
val my_lemma:
  variables ->
  Lemma
    (requires pre_condition variables)
    (ensures  post_condition variables)
```

Our secrecy formula from above already has the right structure to be expressed as an F* lemma. However, we are missing the part about "Alice generated the nonce for Bob". From our model of the Online? protocol, we know that when Alice generates the nonce and sends it to Bob, she stores the nonce together with Bob in a `SentPing` state. We can thus write `state_was_set_some_id tr alice (SentPing {bob; n_a})` to say that Alice generated the nonce $n_A$ for Bob. The property `state_was_set_some_id` just says, that at some point on the trace, Alice stored the state in one of her sessions.

Putting everything together, we can state the first formal secrecy property in DY*:

**Example: Nonce Secrecy for the Online? protocol**

The secrecy of the nonce $n_A$ that Alice generates for Bob, is expresses as the following F* Lemma:

```
val n_a_secrecy:
  tr:trace -> alice:principal -> bob:principal -> n_a:bytes ->
  Lemma
  (requires
     complies_with_online_protocol tr /\
     state_was_set_some_id tr alice (SentPing {bob; n_a})
  )
  (ensures
     attacker_knows tr n_a ==>
     (principal_is_corrupt tr alice \/ principal_is_corrupt tr bob)
  )
```

**Alternative**

It is of course equivalent to move the `attacker_knows` term from the **`ensures`** clause to the **`requires`** clause like this:

```
val n_a_secrecy:
  tr:trace -> alice:principal -> bob:principal -> n_a:bytes ->
  Lemma
  (requires
     complies_with_online_protocol tr /\
     state_was_set_some_id tr alice (SentPing {bob; n_a}) /\
     attacker_knows tr n_a
  )
  (ensures
     principal_is_corrupt tr alice \/ principal_is_corrupt tr bob
  )
```

**Attention**

If you try to copy the security property from above to your DY* model of the Online? protocol, you will get an error (`Identifier not found`), since we didn't define `complies_with_online_protocol` yet. We'll do that only in the next chapter.

We can strengthen this secrecy property for the Online? protocol, by saying that nonces stored in *any* of Alice's states are secret. So far, we only considered those in `SentPing` states, but we have the same secrecy guarantee for nonces stored in her `ReceivedAck` states. So the full secrecy property, we want to show for the Online? protocol is

**Example: Nonce Secrecy for Online? Protocol**

```
val n_a_secrecy:
  tr:trace -> alice:principal -> bob:principal -> n_a:bytes ->
  Lemma
  (requires
     complies_with_online_protocol tr /\ (
       state_was_set_some_id tr alice (SentPing {bob; n_a}) \/
       state_was_set_some_id tr alice (ReceivedAck {bob; n_a} )
     )
  )
  (ensures
     attacker_knows tr n_a ==>
     (principal_is_corrupt tr alice \/ principal_is_corrupt tr bob)
  )
```

**Exercise: Secrecy of the Nonce $n_A$ for the NSL protocol**

Write down the secrecy property for the nonce $n_A$ in the NSL protocol as an F* lemma.

1. What is the intuitive textual description of the secrecy property?

   **Show/Hide Answer**

   For every trace complying with the NSL protocol, the attacker does not get to know the nonce $n_A$ that Alice generated for Bob, as long as both Alice and Bob are honest.

   Or in the stronger version:

   For every trace complying with the NSL protocol, the attacker does not get to know the nonce $n_A$ that Alice stores in one of her states associated with Bob, as long as both Alice and Bob are honest.

2. Write the intuitive description as formula using quantifiers and logical connectives.

   **Show/Hide Answer**

   $\forall$ trace $tr$ $\forall$ Alice, Bob, nonce $n_A$:

   ($tr$ complies with the NSL protocol $\wedge$ Alice stores $n_A$ in one of her states associated with Bob on $tr$) $\implies$

   (attacker knows $n_A$ from $tr$ $\implies$ one of Alice or Bob is corrupt on $tr$)

3. Write the formula as an F* lemma. You can use `complies_with_nsl` as blackbox.

**Show/Hide Answer**

```
val n_a_secrecy:
  tr:trace -> alice:principal -> bob:principal -> n_a:bytes ->
  Lemma
  (requires
    complies_with_nsl tr /\
    ( (state_was_set_some_id tr alice (InitiatorSendingMsg1 bob n_a)) \/
      (exists n_b. state_was_set_some_id tr alice (InitiatorSendingMsg3 bob n_a
↪n_b)))
    ) /\
    attacker_knows tr n_a
  )
  (ensures principal_is_corrupt tr alice \/ principal_is_corrupt tr bob)
```

**Exercise: Secrecy of the Nonce $n_B$ for the NSL protocol**

Write down the secrecy property for the nonce $n_B$ in the NSL protocol as an F* lemma.

**Show/Hide Answer**

```
val n_b_secrecy:
  tr:trace -> alice:principal -> bob:principal -> n_b:bytes ->
  Lemma
  (requires
    complies_with_nsl tr /\
    ( (exists n_a. state_was_set_some_id tr bob (ResponderSendingMsg2 alice n_a n_b))
↪\/
      (exists n_a. state_was_set_some_id tr bob (ResponderReceivedMsg3 alice n_a n_b))
    ) /\
    attacker_knows tr n_b
  )
  (ensures principal_is_corrupt tr alice \/ principal_is_corrupt tr bob)
```

To summarize:

**Remember: General Structure of Secrecy Property**

Secrecy properties usually look like the following

```
val secrecy:
    tr:trace -> alice:principal -> bob:principal -> secret_value:bytes ->
    Lemma
    (requires
      complies_with_protocol tr /\
      ( (state_was_set_some_id tr alice (SomeState1 bob secret_value ...)) \/
        (state_was_set_some_id tr alice (SomeState2 bob secret_value ...)) \/
        ...
      ) /\
      attacker_knows tr secrect_value
    )
```

```
    (ensures principal_is_corrupt tr alice \/ principal_is_corrupt tr bob)
```

Sometimes the above can be strengthened to consider also states *of Bob*. In that case, we add some more `state_was_set_some_id` clauses to the **`requires`**:

```
(state_was_set_some_id tr bob (SomeState alice secret_value ...))
```

## 5.3 Authentication

In contrast to secrecy properties, authentication properties do not talk about attacker knowledge, but about the order of things happening on the trace. They are usually phrased as:

> If y happens, then x must have previously happend.

For example, recall, the responder authentication property for the Online? protocol:

> If Alice at the end of a run believes, she talks with Bob, then this Bob must have been involved in the run, i.e., Bob must have previously sent an acknowledgement as response.

To state this property in DY*, we use *events*, which are a type of trace entries, that we did not discuss so far. Inutitively, those entries log observable protocol actions. Usually, we have one event for every protocol step to log that the respective participant is executing the step.

For example, in the Online? protocol, we will add one event for each of the three protocol steps:

- An "Initiating" event, that Alice triggers, when she starts a new run with some nonce $n_A$ and Bob.
- A "Responding" event, that Bob triggers, when he replies with an acknowledgement $n_A$ to Alice.
- A "Finishing" event, that Alice triggers, when she finishes a run with Bob and the nonce $n_A$, i .e., when she receives the acknowledgement.

With these events, we can then express responder authentication as:

> If Alice triggered a Finishing event, then Bob previously triggered a corresponding Responding event, as long as both Alice and Bob are honest.

As for secrecy, the properties only talk about traces complying with the underlying protocol.

**Example: Responder Authentication for the Online? protocol**

Responder Authentication for the Online? protocol is stated in DY* as the following Lemma:

```
val responder_authentication:
  tr:trace -> ts:timestamp ->
  alice:principal -> bob:principal ->
  n_a:bytes ->
  Lemma
  (requires
     complies_with_online_protocol tr /\
     event_triggered_at tr ts alice (Finishing {alice; bob; n_a})
  )
  (ensures
     principal_is_corrupt tr alice \/ principal_is_corrupt tr bob \/
     event_triggered_before tr ts bob (Responding {alice; bob; n_a})
  )
```

**Exercise: Responder Authentication for the NSL protocol**

Write down the responder authentication property for the NSL protocol as a Lemma. The textual description of the property is:

> If Alice at the end of a run believes to be talking with Bob, then this Bob must indeed be involved in the run.

1. What events do we need to introduce to state the property? Give them speaking names and explain, what they should express/when they are triggered.

   **Show/Hide Answer**

   As for the Online? protocol, we add one event for every protocol step. That is, we have

   - an "Initiating" event, that Alice triggers, when she starts a new run with a nonce $n_A$ and Bob,
   - a "Responding to Message 1" event, that Bob triggers, when he replies to Alice with a second message containing the nonces $n_A$ and $n_B$,
   - a "Responding to a Message 2" event, that Alice triggers, when seh replies to Bob with a third message containing the nonce $n_B$, and
   - a "Finishing" event, that Bob triggers, when he receives the final third message from Alice.

2. Using your events, describe the responder authentication property as a text.

   **Show/Hide Answer**

   If Alice triggers a "Responding to a Message 2" event, then Bob previously triggerd a corresponding "Responding to a Mesage 1" event, as long as both Alice and Bob are honest.

3. Finally, express your property as an F* lemma. (You can again use `complies_with_nsl` as blackbox.)

   **Show/Hide Answer**

```
val responder_authentication:
  tr:trace -> ts:timestamp ->
  alice:principal -> bob:principal -> n_a:bytes -> n_b:bytes ->
  Lemma
  (requires
    complies_with_nsl tr /\
    event_triggered_at tr ts alice (Responding2 {alice; bob; n_a; n_b})
  )
  (ensures
    principal_is_corrupt tr alice \/
    principal_is_corrupt tr bob \/
    event_triggered_before tr ts bob (Responding1 {alice; bob; n_a; n_b})
  )
```

**Exercise: Initiator Authentication for the NSL protocol**

Write down the following initiator authentication property for the NSL protocol as an F* lemma:

> If Bob at the end of a run believes to be talking with Alice, then this Alice must indeed be involved in the run.

**Show/Hide Answer**

```
val initiator_authentication:
```

```
  tr:trace -> ts:timestamp ->
  alice:principal -> bob:principal -> n_a:bytes -> n_b:bytes ->
  Lemma
  (requires
    complies_with_nsl tr /\
    event_triggered_at tr ts bob (Finishing {alice; bob; n_a; n_b})
  )
  (ensures
    principal_is_corrupt tr alice \/
    principal_is_corrupt tr bob \/
    event_triggered_before tr ts alice (Responding2 {alice; bob; n_a; n_b})
  )
```

**Remember: General Structure of Authentication Property**

Authentication properties in DY* usually look like the following:

```
val bob_authentication:
  tr:trace -> ts:timestamp ->
  alice:principal -> bob:principal ->
  Lemma
  (requires
    complies_with_protocol tr /\
    event_triggered_at tr ts alice (Event2 alice bob ...)
  )
  (ensures
     principal_is_corrupt tr alice \/
     principal_is_corrupt tr bob \/
     event_triggered_before tr ts bob (Event1 alice bob ...)
  )
```

## 5.4 How To: State Security Properties in DY*

**How-To: Stating Security Properties in DY***

- Security properties in DY* are *trace-based* and as such talk about traces that *comply* with the protocol of interest.
- The attacker can read any message on the trace and contents of states he corrupted. From those values, he can build more terms with cryptographic functions. The *attacker knowledge* is the set of all terms the attacker can construct.
- *Secrecy* properties talk about values that stay secret (unknown to the attacker) during runs of a protocol. They look like:

```
val secrecy:
    tr:trace -> alice:principal -> bob:principal -> secret_value:bytes ->
    Lemma
    (requires
      complies_with_protocol tr /\
      ( (state_was_set_some_id tr alice (SomeState1 bob secret_value ...)) \/
        (state_was_set_some_id tr alice (SomeState2 bob secret_value ...)) \/
        ...
      ) /\
      attacker_knows tr secrect_value
    )
    (ensures principal_is_corrupt tr alice \/ principal_is_corrupt tr bob)
```

- *Authentication* properties talk about a sequence of actions. They are expressed with *events*. We usually have one event per protocol step. The authentication property then looks like:

```
val bob_authentication:
  tr:trace -> ts:timestamp ->
  alice:principal -> bob:principal ->
  Lemma
  (requires
    complies_with_protocol tr /\
    event_triggered_at tr ts alice (Event2 alice bob ...)
  )
  (ensures
     principal_is_corrupt tr alice \/
     principal_is_corrupt tr bob \/
     event_triggered_before tr ts bob (Event1 alice bob ...)
  )
```

CHAPTER

SIX

# PROVING SECURITY PROPERTIES

## 6.1 The Proof

- two independent parts of the proof:
  - trace invariants ==> security property
  - steps maintain invariants

## 6.2 Labeling System

- only high-level idea of "intended readers" should suffice
- recursive definition of labels starting from Nonces
- intuition of `l1 can_flow l2` as l2 is more secret than l1

## 6.3 Trace Invariants

The idea: specify "trace complies with protocol"

- crypto invariants
- state pred
- event pred

## 6.4 Proof of Secrecy for Online? Protocol

- invariants + relevant changes to code (nonce label)

## 6.5 Proof of Responder Authentication for Online? Protocol

- introducing events
- one event per protocol step / per state
- state predicate refers to event predicate

## 6.6 Proving Security Properties for NSL

## 6.7 How To: Prove Security Properties in DY*