Project Documentation - Onion Module

Group 10 - Leon Beckmann, Florian Freund

November 3, 2021

Structure

1 Introduction

During the Peer-to-Peer Systems and Security course at the Computer Science department of the Technical University of Munich, the goal was to create an anonymous and unobservable VoIP application on top of a structured P2P network. The whole application has been divided into different sub-tasks, i.e. layered modules, similar to the OSI layers. The Call Management Module (CM) is responsible for the call management, the user interface and the interface with the sound subsystem of the OS. The Onion Module takes the commands and forwards the traffic from and to the CM and builds secure onion tunnels. These onion tunnels are routed via other, random selected, peers, which are sampled via the Random-Peer-Sampling Module (RPS). RPS knows the peers from the Gossip Module, which is responsible for spreading information of peers in the network, and then builds a probability distribution based on the network size estimation for random sampling of the peers. The network size is estimated by the Network-Size-Estimation Module (NSE). The Distributed-Hash-Table Module (DHT) forms a structured overlay of the network via a distributed hash-table.

This project documentation is about the Onion module, which has been implemented fully asynchronously in Rust using the Tokio library. It is structured as follows: Section 2 describes the full architecture and design of the Onion module. Afterwards, section 3 analysis whether and how the different security goals are reached. The software documentation can be found in section 4, which includes the configuration via an INI file, build dependencies and a guide how to test and run the module. Future work is listed in section 5.

2 Module Architecture

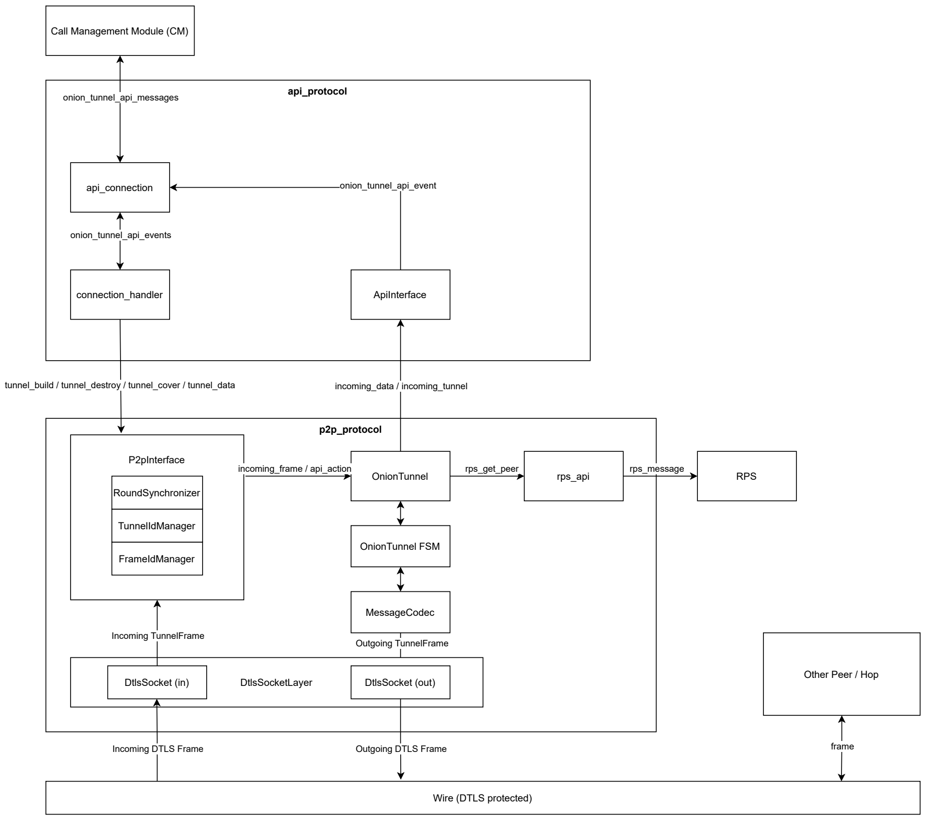

Figure 1: Logical structure of the Onion Module

In figure 1 is given a general overview of the Onion Module architecture. It is divided into the API protocol package and the P2P protocol package. The API protocol is the interface between the Call Management Module (CM) and the P2P protocol and is responsible for handling CM/API connections, as well as for delegating CM messages to the P2P protocol and passing incoming events from the P2P protocol to the CM. The P2P protocol takes and executes the requests from the API protocol, is listening for incoming connections and is responsible for the onion tunnel management and the round synchronization. The protocols own weak references to the interface of the other protocol (*ApiInterface* and *P2pInterface*) for communication. In the following, both protocols and their components are described in detail.

2.1 API Protocol Package

The API protocol is used as the interface between the Call Management Module (CM) and the Onion Module.

2.1.1 CM Interface

Communication between the CM and the API protocol is done via tunnel messages, which are described in the following:

The **ONION TUNNEL BUILD** message (see figure 2) is used by the CM to request a new tunnel to a given destination in the next round. It is identified by the *ONION TUNNEL BUILD* message type. It contains the onion port and the onion IP address of the destination peer, as well as the public identity hostkey of the destination peer in DER format. The **V** flag is cleared if the destination IP address is in IPv4 format and set if it is in IPv6 format.

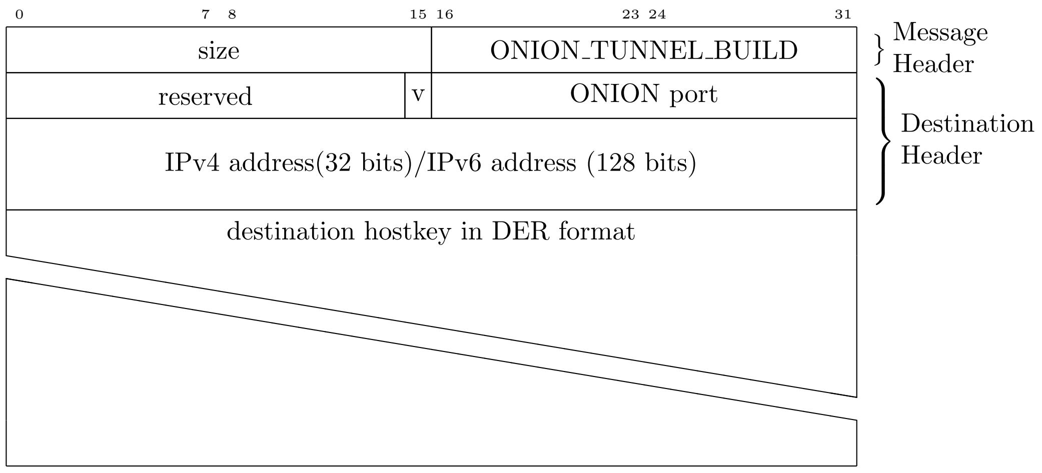

Figure 2: ONION TUNNEL BUILD message

The **ONION TUNNEL READY** message (see figure 3) is used by the API protocol to signal the CM that a new caller tunnel is built, caused by an earlier request. It is identified by the *ONION TUNNEL READY* message type. It contains the new 32-bit tunnel ID associated with the new tunnel and the hostkey of the callee in DER format.

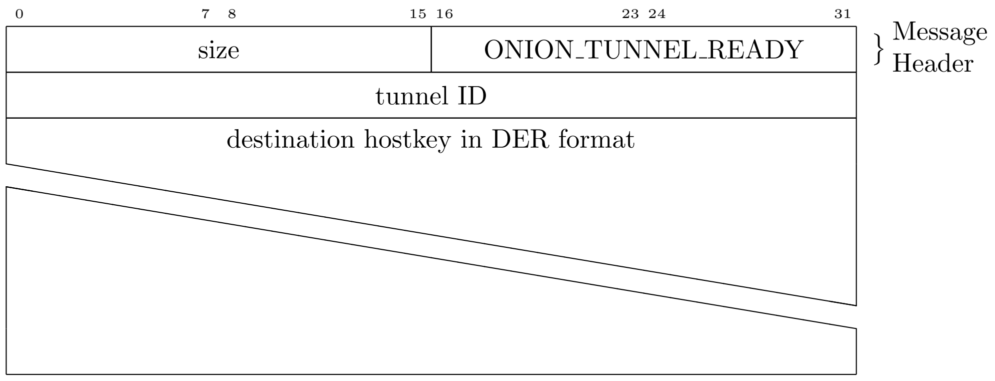

Figure 3: ONION TUNNEL READY message

The **ONION TUNNEL INCOMING** message (see figure 4) is used by the API protocol to signal the CM that a new callee tunnel is built, caused by another peer. It is sent to all open API connections. It is identified by the *ONION TUNNEL INCOMING* message type. It contains the new 32-bit tunnel ID associated with the new tunnel.

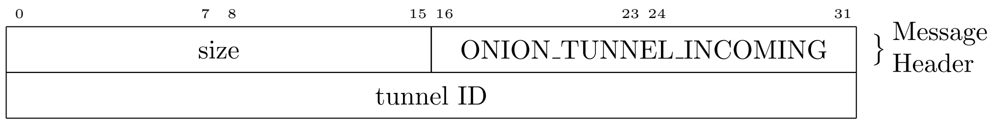

Figure 4: ONION TUNNEL INCOMING message

The **ONION TUNNEL DESTROY** message (see figure 5) is used by the CM to unsubscribe a tunnel, identified by its tunnel ID. It is identified by the *ONION TUNNEL DESTROY* message type. It contains the 32-bit tunnel ID associated with the tunnel, from which the connection want so unsubscribe. When all registered connections have destroyed the reference on a specific tunnel, the tunnel will be closed.

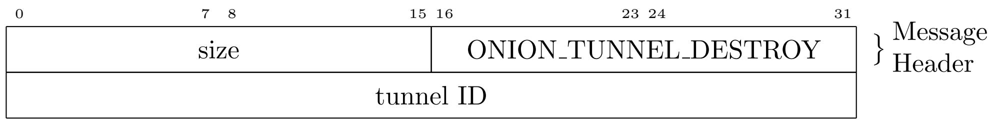

Figure 5: ONION TUNNEL DESTROY message

The **ONION TUNNEL DATA** message (see figure 6) is used by the CM to send new data via a tunnel and by the API protocol to forward incoming data to the CM. It is identified by the *ONION TUNNEL DATA* message type. It contains the 32-bit tunnel ID associated with the tunnel that is/should be used for the data exchange and the data itself. Note that no guarantee is given whether outgoing traffic will reach its destination.

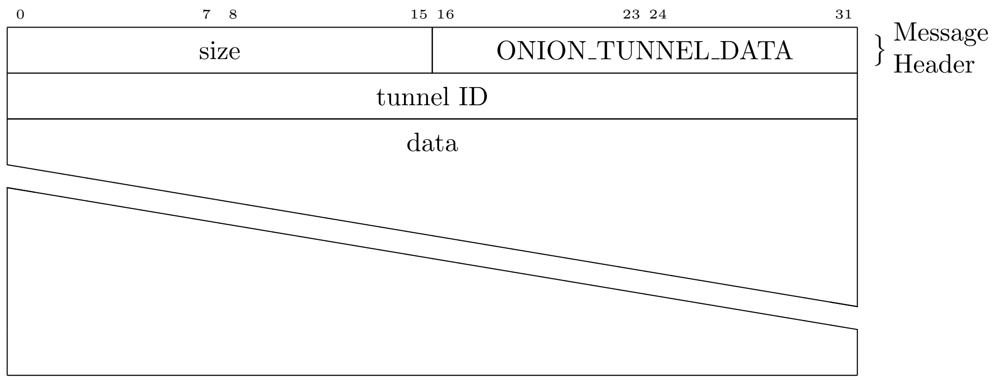

Figure 6: ONION TUNNEL DATA message

The **ONION ERROR** message (see figure 7) is used by the API protocol to signal the CM that an error has occurred. It is identified by the *ONION ERROR* message type. It contains the 32-bit tunnel ID and the 16-bit request type to indicate to which request and to which tunnel the error refers to.

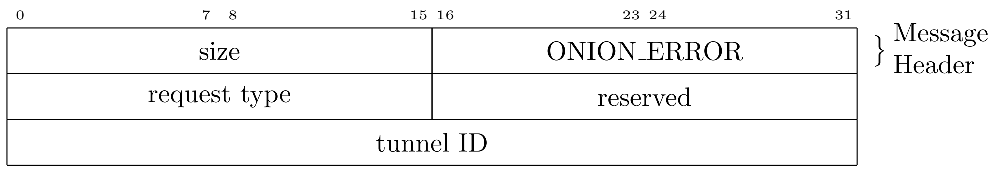

Figure 7: ONION ERROR message

The **ONION COVER** message (see figure 8) is used by the CM to send cover traffic, simulated as real VoIP traffic. It is identified by the *ONION COVER* message type. It contains the 16-bit *cover_size* that signals how may random bytes should be sent via the current tunnel. Note that the specification requires that no such message must be sent when there is only a caller tunnel active, which means that a caller tunnel cannot send cover traffic. However, we think that it might be desirable when calls with less data exchange will be extended by additional cover traffic to have similar characteristics than calls with a higher number of packets sent. Therefore, the design of the module also allows cover traffic messages for caller tunnels. Nevertheless, due to the specification, these requests are blocked for now at a higher level within the onion tunnel.

0 … 15	16 … 31	
size	ONION_COVER	} Message Header
cover_size	reserved	

Figure 8: ONION COVER message

2.1.2 Onion Message Types

The 16-bit message types, mentioned in the previous part, are defined by the project specification as follows:

- **ONION TUNNEL BUILD** = 560
- **ONION TUNNEL READY** = 561
- **ONION TUNNEL INCOMING** = 562
- **ONION TUNNEL DESTROY** = 563
- **ONION TUNNEL DATA** = 564
- **ONION ERROR** = 565
- **ONION COVER** = 566

2.1.3 Internal Logic

The tunnel messages from the previous part are parsed to/from internal (incoming and outgoing) events. The API interface is started by calling its *listen* method, which takes the weak reference for the *P2pInterface* and the API socket address of the onion layer from the configuration file. On the socket address, a TCP listener is started, that is waiting for incoming connections from the CM. For each new request, a new API *Connection* is created and handled using a *connection_handler*. This handler first registers the new connection at the connection registry and then handles new incoming events from the CM using a *event_handler* per event. The *event_handler* might return an outgoing event, which is the response on an earlier request and must be sent back to the CM via the open API connection. When the connection has been closed, the *connection_handler* unregisters the connection from the registry and terminates. Simply said, the API connection abstracts the TCP stream and the tunnel message parsing, such that it can be used for reading incoming events and writing outgoing events. Such API *Connections* will be registered to incoming and outgoing caller tunnels in the P2P protocol as connection listeners.

2.2 P2P Protocol Package

The p2p protocol is responsible for rounds and secure onion tunnels and is communicating with the CM via the API protocol. When a peer starts, the *listen* method of the *P2pInterface* is called to listen on incoming DTLS packets. The DTLS connections to other peers are managed within the *DtlsSocketLayer*. A *RoundSynchronizer* is initialized and responsible for the synchronized onion tunnel creation (exactly one at a time) and the tunnel destruction at the end of a round. The *Tunnel-ID-Manager* and *Frame-ID-Manager* are responsible for passing incoming frames to the corresponding tunnel and for tunnel management. The *OnionTunnel* is the secure channel of a call and consists of multiple components, such as a finite state machine to handle different events in different states of the call and a *MessageCodec*, which is responsible for parsing and sending tunnel messages. The *rps_api* is used to communicate with the Random-Peer-Sampling (RPS) Module when intermediate peers have to be selected for a new tunnel. All these components are described in more detail in the following.

2.2.1 Round Synchronizer

The Onion Module makes use of a round synchronizer that splits the time into a sequence of equal periods, so called rounds. It is assumed that the round duration and starting time is arbitrary but fixed between all peers. To prevent re-identification attacks based on the number of outgoing tunnels, each peer must ensure that there is exactly the same number of outgoing tunnels per round per peer (in our case: one). This outgoing tunnel could either be a cover-traffic-only tunnel or a caller tunnel. In general, there are three different possibilities:

1. There exists an active caller tunnel in round T that is not downgraded or terminated until the beginning of the build window for round $T+1$. Then the new tunnel in round $T+1$ is used as a rebuilt of the caller tunnel of round T to make the call persistent over multiple rounds.

2. There is no active caller tunnel of round T available anymore until the beginning of the build window for round $T+1$, but the CM Module has requested a new caller tunnel during the registration window for round $T+1$. Then the new tunnel in round $T+1$ is used as a caller tunnel for this request.

3. There is neither an active caller tunnel nor a new caller tunnel request until the beginning of the build window for round $T+1$. Then the new tunnel in round $T+1$ is used only for cover traffic.

A detailed overview of the round architecture is given in figure 9. The split into registration window and build window ensures that all the three possibilities for tunnel constructions are performed at a comparable time during the build

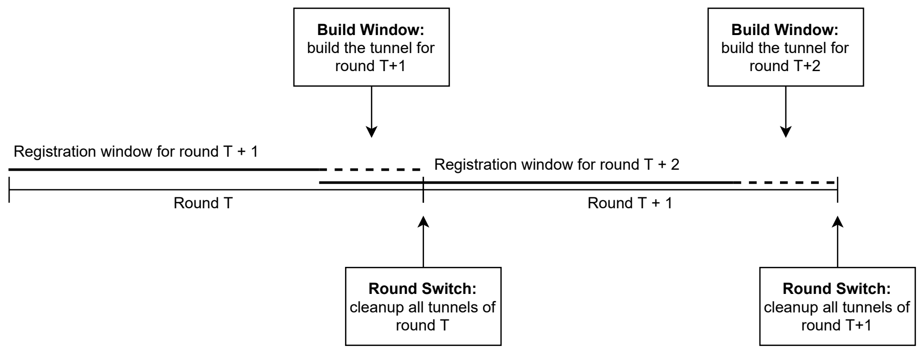
Build Window: build the tunnel for round T+1
Build Window: build the tunnel for round T+2
Registration window for round T + 1
Registration window for round T + 2
Round T
Round T + 1
Round Switch: cleanup all tunnels of round T
Round Switch: cleanup all tunnels of round T+1

Figure 9: Round Architecture

window, such that no attacker could identify the type of tunnel due to its construction time.
Once a tunnel is built it is considered as active. In the case of a tunnel update (i.e. a caller tunnel of round T is rebuilt for round $T+1$), the communication is transparently switched over via links. All old tunnels (incoming and outgoing) stay until the end of a round and are then terminated via the round switch. Thus, for a short amount of time during the build window, each peer has exactly two outgoing tunnels. This is required to avoid packet losts for updated tunnels: The outgoing traffic of the caller and the callee is already transferred via the new tunnel, but there might exist traffic, sent before the switch over, that has not reached the other participant before its switch over. Dependent on the configurable duration of the build window, the old tunnel still exists for a short time and thereby this traffic can still be received. This does also hold for cover traffic, that has been sent via the old tunnel of round T before the switch over.

Remark. *That there will exist two outgoing tunnel at the end of the build window per peer is fine, since the requirement to prevent re-identification attacks based on the number of outgoing tunnels only expects that the number of outgoing tunnels per peer is equal at every time, which still holds.*

Remark. *It could happen that the Call Management Module requests multiple new caller tunnels or a new caller tunnel while a previous call is still active and thus updated. In this case, all the additional new tunnel requests are answered with a ModuleEngaged error.*

Remark. *A tunnel must not be destroyed before the end of a round to ensure that caller tunnels and cover tunnels have the same lifetime and cannot be distinguished due to their destruction time. Therefore, a cover tunnel cannot be destructed, while a caller tunnel can only be downgraded to a cover tunnel at the end of a call, such that it is still available until the end of the round for passing* *cover traffic. (This does not include cleanup of broken connections due to peer shutdowns, which could occur for both, caller and cover tunnels.)*

2.2.2 Tunnel IDs and Tunnel-ID-Manager

Within the Onion Module, onion tunnels are identified by its 32-bit tunnel ID. This tunnel ID is used for mapping CM requests and incoming frames (via frame IDs) to the corresponding tunnel. Further, the tunnel ID is not shared between peers, such that each peer has its own ID counter and does not know the ID of a tunnel used at the participant peer.
The Tunnel-ID-Manager is synchronized behind a Read-Write-Lock to ensure exclusive write access. It holds a hash-map that acts as a tunnel registry and owns the onion tunnels, keyed by their tunnel ID. The Tunnel-ID-Manager is then responsible for managing the tunnels, i.e. inserting new tunnels after construction, removing old tunnels after destruction and accessing available tunnels. Further, the Tunnel-ID-Manager holds hash-maps for links and reverse links for updated caller tunnels: Once a caller tunnel is created and connected, the CM receives a *TunnelReady* message with the corresponding tunnel ID. During the update/rebuild of a caller tunnel via the round synchronizer, the new tunnel used for the call will have another tunnel ID than the previous tunnel. To ensure that the CM is still able to communicate with the callee via the old tunnel ID (the only ID known by the CM), the old tunnel ID is mapped to the new tunnel ID via the Tunnel-ID-Manager's *links* hash-map. The *reverse-links* hash-map is used for mapping the new tunnel ID of a tunnel to the old tunnel ID, known by the CM, to pass incoming data to the CM. These links are updated when a tunnel is rebuilt multiple times, such that the connection between active tunnels and the original tunnel ID is always given.

2.2.3 Frame IDs and Frame-ID-Manager

To handle an incoming data frame, it has to be mapped to its corresponding tunnel ID. However, there are some challenges which have to be respected:

1. After the ECDHE parameters have been exchanged during the handshake between the initiator and another peer, the data is sent encrypted to ensure confidentiality.

2. Frames must ensure to have a fixed size to prevent packet-size analysis attacks. This is why there cannot be any kind of additional encrypted routing information for an intermediate hop, provided by the initiator, when layered onion encryption is used. Otherwise, this encrypted information, which is required for each hop, must either be removed after being processed at a hop or a hop needs to know its position within the connection, which is not allowed for privacy reasons. However, if the information is removed, the packet size is reduced and packet size analysis would in turn reveal the position of a node in the connection.

3. There must not be any trackable information included in the frame among several peers.

4. The sender address of the incoming frame cannot be used for identifying the corresponding tunnel, since there might be multiple tunnels routed this way. Thus, the sender address might not be unique.

The solution is presented in figure 10 for forward communication and in figure 11 for backward communication, where the boxes below the peers list the frame IDs known at the specific peer. When a tunnel is constructed at a peer, each peer reserves a locally unique (unique per peer) *Forward Frame ID* and a locally unique *Backward Frame ID* via its Frame-ID-Manager. These IDs are used for mapping incoming forward packets and incoming backward packets to the corresponding tunnel IDs. During the handshake protocol, the frame IDs are exchanged such that each peer holds the *Forward Frame ID* of its successor and the *Backward Frame ID* of its predecessor (see figure 12). When the initiator now wants to send a frame to the target peer via two intermediate hops, the initiator sets the *Forward Frame ID* of the first hop. The first hop can then map the frame via the Frame-ID-Manager's hash-map to the corresponding tunnel ID, which is then mapped via the Tunnel-ID-Manager to the corresponding tunnel, as described above. Then the first hop replaces the old ID of the frame by the *Forward Frame ID* of the second hop and forwards the packet. The second hop acts the same and replaces the ID of the frame by the *Forward Frame ID* of the target peer. For the backward direction the process is exactly the same, but the *Backward Frame ID* is used.

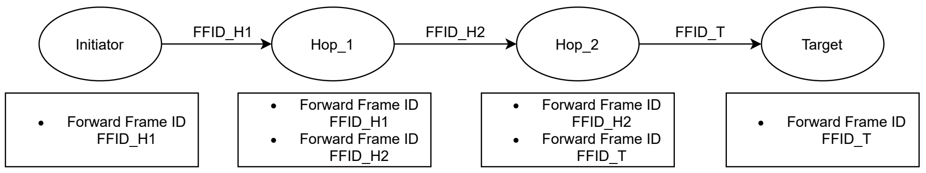
Initiator
FFID_H1
Hop_1
FFID_H2
Hop_2
FFID_T
Target
• Forward Frame ID FFID_H1
• Forward Frame ID FFID_H1
• Forward Frame ID FFID_H2
• Forward Frame ID FFID_H2
• Forward Frame ID FFID_T
• Forward Frame ID FFID_T

Figure 10: Forward communication via Forward Frame IDs

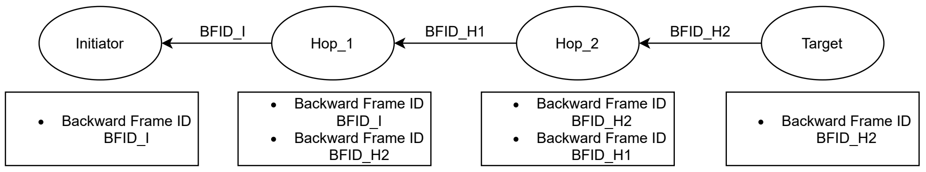
Initiator
BFID_I
Hop_1
BFID_H1
Hop_2
BFID_H2
Target
• Backward Frame ID BFID_I
• Backward Frame ID BFID_I
• Backward Frame ID BFID_H2
• Backward Frame ID BFID_H2
• Backward Frame ID BFID_H1
• Backward Frame ID BFID_H2

Figure 11: Backward communication via Backward Frame IDs

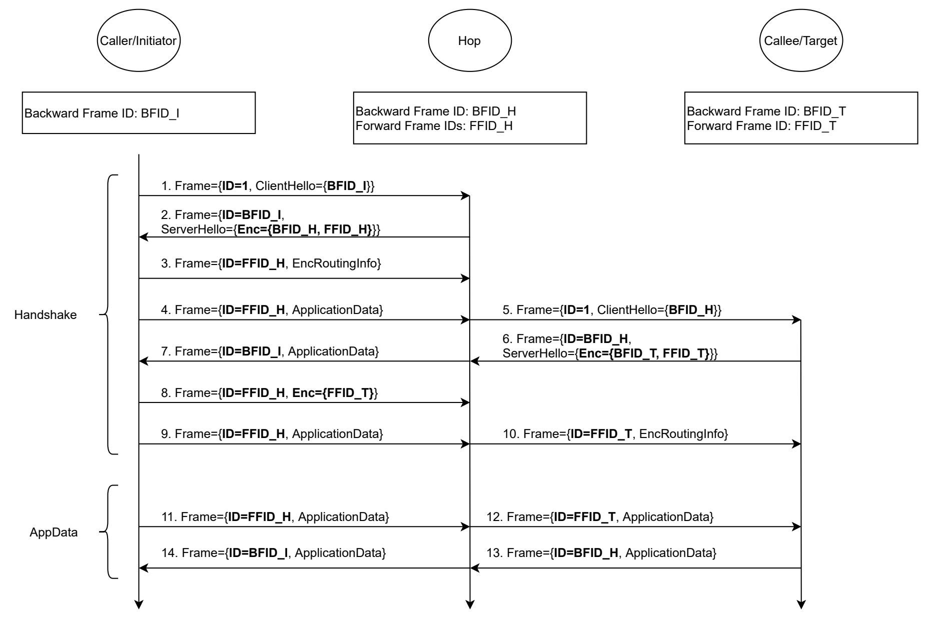
Caller/Initiator
Hop
Callee/Target
Backward Frame ID: BFID_I
Backward Frame ID: BFID_H
Forward Frame IDs: FFID_H
Backward Frame ID: BFID_T
Forward Frame ID: FFID_T
Handshake
1. Frame={ID=1, ClientHello={BFID_I}}
2. Frame={ID=BFID_I, ServerHello={Enc={BFID_H, FFID_H}}}
3. Frame={ID=FFID_H, EncRoutingInfo}
4. Frame={ID=FFID_H, ApplicationData}
5. Frame={ID=1, ClientHello={BFID_H}}
6. Frame={ID=BFID_H, ServerHello={Enc={BFID_T, FFID_T}}}
7. Frame={ID=BFID_I, ApplicationData}
8. Frame={ID=FFID_H, Enc={FFID_T}}
9. Frame={ID=FFID_H, ApplicationData}
10. Frame={ID=FFID_T, EncRoutingInfo}
AppData
11. Frame={ID=FFID_H, ApplicationData}
12. Frame={ID=FFID_T, ApplicationData}
14. Frame={ID=BFID_I, ApplicationData}
13. Frame={ID=BFID_H, ApplicationData}

Figure 12: Frame ID exchange during the handshake

Remark. *To recognize new tunnel requests, the ClientHello frame contains the Forward Frame ID 1.*

Someone might have the thought that a global sniffer could observe the number of packets per frame ID, and thereby uncover entire tunnels, since the number of packets per tunnel might be a unique fingerprint, depending on the implementation. This is prevented by having authenticated secure channels between each two peers, such that the tunnel frames with their frame IDs are sent encrypted.

Remark. *The Forward Frame ID of the target is stored at the initiator and is used for tunnel updates. Here, the initiator references the old Forward Frame ID via the RoutingInformation handshake message addressed to the target. The target can then identify the corresponding tunnel that is updated by the new connection.*

2.2.4 Onion Tunnels

The calls via the Onion Module are represented by onion tunnels, more precisely caller tunnels and cover-traffic-only tunnels. As described in section 2.2.1, per round there is exactly one outgoing tunnel per peer. Caller tunnels allow both, sending user data and cover traffic via the tunnels to the callee. They can be downgraded to cover-traffic-only tunnels, which does not allow sending user

data anymore. Each tunnel is destructed at the end of the round, but must not be terminated during a round for privacy reasons, except the connection is broken. Further, a caller tunnel has a set of API connection listeners, which have subscribed to the call to get notifications about incoming messages, while the incoming data of cover-traffic-only tunnels is simply be mirrored back to the initiator. A caller tunnel can be updated, i.e. rebuilt, to make the call persistent for the next round when it has not been downgraded until the beginning of the build window for the next round.
The internal logic of such an onion tunnel is designed via a finite state machine (FSM), which consists of the main FSM (see figure 13), describing the tunnel state, and the outsourced handshake FSM (see figure 14), which describes a sub-state of the main FSM. Using a finite state machine eliminates unexpected or non-handled cases, since for every state it is clearly defined how to react on every event. The communication with the FSM, as well as the communication between the main and the handshake FSM is done via Rust's multi-producer-single-consumer channels to ensure LIFO ordering and synchronization.

Main FSM

The main FSM supports the following events:

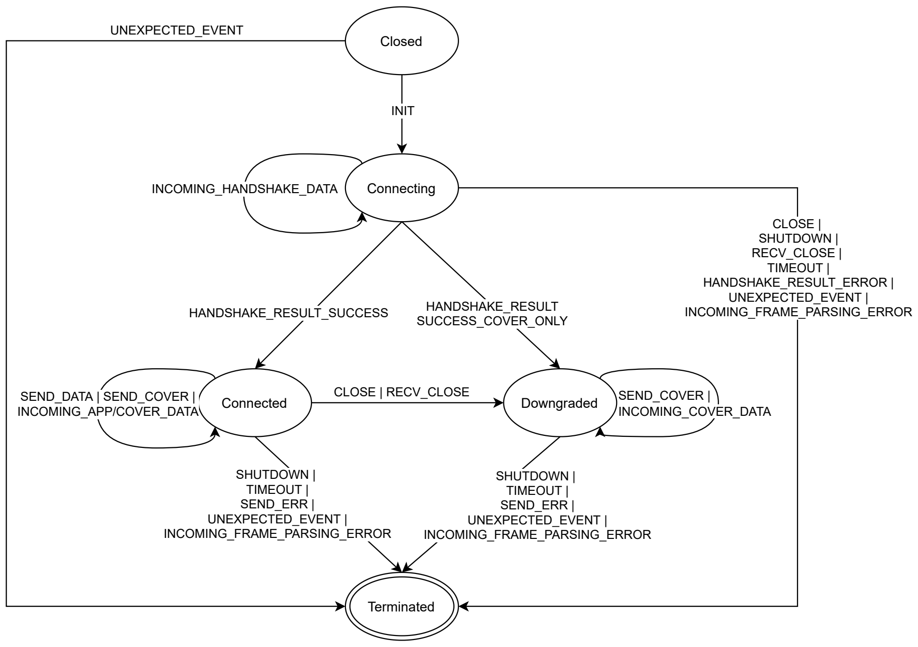
UNEXPECTED_EVENT
Closed
INIT
INCOMING_HANDSHAKE_DATA
Connecting
CLOSE | SHUTDOWN | RECV_CLOSE | TIMEOUT | HANDSHAKE_RESULT_ERROR | UNEXPECTED_EVENT | INCOMING_FRAME_PARSING_ERROR
HANDSHAKE_RESULT_SUCCESS
HANDSHAKE_RESULT SUCCESS_COVER_ONLY
SEND_DATA | SEND_COVER | INCOMING_APP/COVER_DATA
Connected
CLOSE | RECV_CLOSE
Downgraded
SEND_COVER | INCOMING_COVER_DATA
SHUTDOWN | TIMEOUT | SEND_ERR | UNEXPECTED_EVENT | INCOMING_FRAME_PARSING_ERROR
SHUTDOWN | TIMEOUT | SEND_ERR | UNEXPECTED_EVENT | INCOMING_FRAME_PARSING_ERROR
Terminated

Figure 13: The main FSM of an onion tunnel

1. **INIT**: Trigger a handshake and create the handshake FSM(s).

2. **INCOMING_HANDSHAKE_DATA**: Received a handshake message.
3. **CLOSE**: Used for downgrading a caller tunnel and send a close message to the other peer.
4. **RECV_CLOSE**: Received a close message from the other peer, which signals a tunnel downgrade.
5. **SHUTDOWN**: Destruct the tunnel, triggered after round finish.
6. **TIMEOUT**: A timeout that is triggered after receiving no messages for a configurable duration, which signals that the tunnel might be broken.
7. **HANDSHAKE_RESULT_OK(is_cover_tunnel)**: The handshake was successful and the value of the is_cover_tunnel signals if the new tunnel should be a downgraded or a connected tunnel.
8. **HANDSHAKE_RESULT_ERROR**: The handshake has failed.
9. **SEND_DATA**: Send CM data to the other endpoint.
10. **SEND_COVER**: Send cover traffic to the other endpoint.
11. **INCOMING_DATA**: Received incoming data from the other endpoint.
12. **INCOMING_COVER**: Received incoming cover data from the other endpoint.
13. **PARSING_ERROR**: Cannot parse the incoming tunnel frame to handshake or application data.
14. **SEND_ERROR**: Cannot send data or cover traffic.

In figure 13, the *UNEXPECTED_EVENT* means receiving any of the events that is not expected in this state, e.g. the *INIT* event in the *CONNECTING* state. The main FSM consists of the following five states:

1. **CLOSED**: The initial state after creation. It only expects the *INIT* event, which triggers an initialization action to create a handshake FSM and then goes into the *CONNECTING* state on success. Each other event leads to termination.
2. **CONNECTING**: The state in which the handshake FSM is active. When receiving a *HANDSHAKE_RESULT_SUCCESS(is_cover_tunnel)* event from the handshake FSM, the main FSM goes either into the *CONNECTED* state or into the *DOWNGRADED* state, depending on the value of the *is_cover_tunnel* variable. Incoming frames that can be successfully parsed to handshake data are forwarded to the handshake FSM, while the main FSM stays in the *CONNECTING* state. All other events lead to termination.

3. **CONNECTED**: The state, in which the caller tunnel is established and communication, i.e.sending and receiving data and cover data, is enabled. When the tunnel manager closes the tunnel via a *CLOSE* event or a *RECV_CLOSE* event is received from the communicating peer, the caller tunnel is downgraded to a cover-traffic-only tunnel. All other events lead to termination.
4. **DOWNGRADED**: The state, in which the cover-traffic-only tunnel is established and communication, i.e. sending and receiving cover data, is enabled. All other events lead to termination.
5. **TERMINATED**: The final state without any outgoing transitions.

Due to the differences of the initialization actions of the initiator of a tunnel and the target (hop or real target), the FSM is implemented as a trait with default implementations for the shared functionality, while the peer-specific functions are implemented by the specific implementations. The target FSM simply creates the handshake FSM, runs it, and builds its communication channel between the main FSM and handshake FSM. In contrast, the FSM of the initiator loops over all hops and the target peer and stays in the state *CONNECTING* until the whole tunnel is established. For this, the handshake result for each handshake with a peer is hooked and caught and new handshake FSMs are created sequentially for each next hop. The underlying communication layer (MessageCodec) is updated after each successful handshake to an intermediate hop, such that the data to the next hop are tunneled through the partially established tunnels.

Remark. *As already noted in section 2.1.1, the specification requires not sending cover messages in connected tunnels. However, we do not see the sense of this requirement, since the CM Module is fully responsible for sending both, cover and real traffic and we think it might be desirable to send cover traffic via real tunnels to balance tunnels with much and little traffic. This is why the design of the state machine allows sending cover traffic in caller tunnels. Nevertheless, this future is disabled by blocking such requests directly before they are forwarded to the FSM.*

Handshake FSM

The handshake FSM is used for both, the peer initiating the handshake and the peer that is acting as the server. However, different initialization states are used. It consists of the following six states:

1. **START**: The initialization state for the client side. It only expects the *INIT* event, which triggers the sending of the ClientHello message and the switch into the *WAIT_FOR_SERVER_HELLO* state. On any error or any other event, the handshake FSM goes into the *ERROR* state.

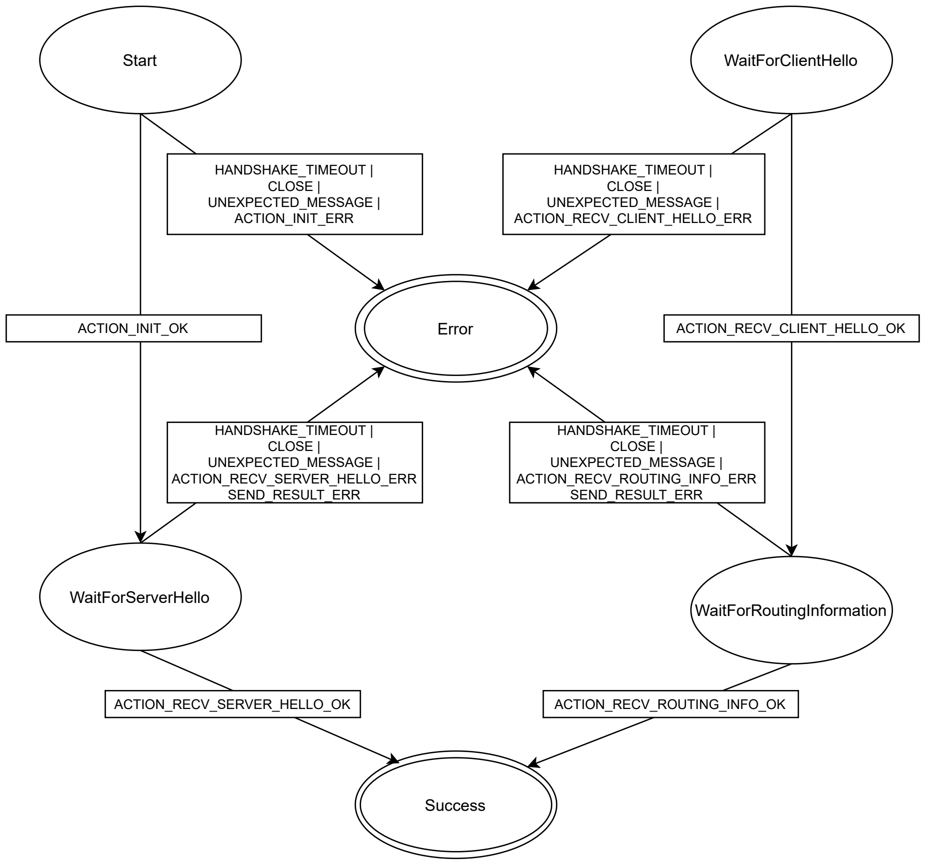
Start
WaitForClientHello
HANDSHAKE_TIMEOUT | CLOSE | UNEXPECTED_MESSAGE | ACTION_INIT_ERR
HANDSHAKE_TIMEOUT | CLOSE | UNEXPECTED_MESSAGE | ACTION_RECV_CLIENT_HELLO_ERR
ACTION_INIT_OK
Error
ACTION_RECV_CLIENT_HELLO_OK
HANDSHAKE_TIMEOUT | CLOSE | UNEXPECTED_MESSAGE | ACTION_RECV_SERVER_HELLO_ERR SEND_RESULT_ERR
HANDSHAKE_TIMEOUT | CLOSE | UNEXPECTED_MESSAGE | ACTION_RECV_ROUTING_INFO_ERR SEND_RESULT_ERR
WaitForServerHello
WaitForRoutingInformation
ACTION_RECV_SERVER_HELLO_OK
ACTION_RECV_ROUTING_INFO_OK
Success

Figure 14: The handshake fsm of an onion tunnel

2. **WAIT_FOR_CLIENT_HELLO**: The initialization state for the server side. It expects the incoming ClientHello message, which triggers the sending of the ServerHello message and the switch into the *WAIT_FOR_ROUTING_INFORMATION* state. On any error (e.g. signing error, sending error, key exchange error), handshake timeout or an unexpected event, the handshake FSM goes into the *ERROR* state.

3. **WAIT_FOR_SERVER_HELLO**: This state is used on client side. It expects the ServerHello message, which triggers the creation of the encrypted RoutingInformation accordingly the P2P handshake protocol. Afterwards, the handshake FSM switches into the *SUCCESS* state. On any error (e.g. signature error, sending error, key exchange error), handshake timeout or an unexpected event, the handshake FSM goes into the *ERROR* state.

4. **WAIT_FOR_ROUTING_INFORMATION**: This state is used on server

side. It expects the encrypted RoutingInformation message, which triggers the processing of the RoutingInformation accordingly the P2P handshake protocol. Afterwards, the handshake FSM switches into the *SUCCESS* state. On any error, handshake timeout or an unexpected event, the handshake FSM goes into the *ERROR* state.

5. **SUCCESS**: The final state that, on entry, triggers the sending of the *HANDSHAKE_RESULT_OK(is_cover_tunnel)* event to the main FSM.

6. **ERROR**: The final state that, on entry, triggers the sending of the *HANDSHAKE_RESULT_ERROR* event to the main FSM.

2.2.5 Message formats

The Onion Module makes use of protobuf messages (version 3)[1] for communication between peers. Each message sent within the Onion Module is a 1024-byte *TunnelFrame*, which contains the 32-bit *FrameID* for mapping the frame to the receiving tunnel, the 16 byte IV used for encryption and the (encrypted) data. Each frame is built within the MessageCodec and it is ensured to be 1024 byte. When the endpoint receives a frame, after all intermediate hops have applied the layered encryption/decryption, the *data* field within the *TunnelFrame* consists of a prefixed 16-byte authentication tag, used for the end-to-end integrity protection of AES-GCM, and the encrypted *FrameData* message. Thus, first the endpoint has to decrypt the whole *data* field using AES-GCM to check the authentication tag and to decrypt the remaining data, which can then be parsed into the *FrameData* type. Such a *FrameData* format contains the *data_size*, which signals where to split the *data* bytes into the serialized *FrameDataType* message and the padding. This has to happen here, since the padding for fixed packet length is a secret information and must therefore be encrypted with the data. Such a *FrameDataType* then holds one of the following:

1. *HandshakeData*: *ClientHello*, *ServerHello* or *RoutingInformation*, as used in the handshake protocol.

2. *ApplicationData*: Used for connected or downgraded tunnels and could either be raw bytes, a *CoverTraffic* message or a *Close* message.

3. *32-bit Forward FrameID*: Used for telling an intermediate hop how to address the next hop (see step 8 in figure 12).

The full protobuf file is shown in the following listing:

```
syntax = "proto3";

message TunnelFrame {
  // an identifier for identifying the corresponding tunnel of↩
     this frame
```

[1]https://developers.google.com/protocol-buffers/docs/proto3

```
  fixed32 frame_id = 1;
  // random, unpredictable iv used for encryption
  bytes iv = 2;
  // auth_tag | FrameData as bytes (encrypted)
  bytes data = 3;
}

message FrameData {
  // size of data, used to split data and padding
  fixed32 data_size = 1;
  // data (FrameDataType), appended by padding
  bytes data = 3;
}

message FrameDataType {
  oneof message {
    HandshakeData handshake_data = 1;
    ApplicationData app_data = 2;
    // forward frame id for telling intermediate hop how to ↩
    address next hop
    uint32 forward_frame_id = 3;
  }
}

message HandshakeData {
  oneof message {
    // Client Hello is the first handshake message
    ClientHello client_hello = 1;
    // Server Hello is the second handshake message
    ServerHello server_hello = 2;
    // Routing Information is the third handshake message
    RoutingInformation routing = 3;
  }
}

message ClientHello {
  // public frame id used for communicating with the caller
  uint32 backward_frame_id = 1;
  // public ECDHE parameter
  bytes ecdh_public_key = 2;
}

message ServerHello {
  // public ECDHE parameter
  bytes ecdh_public_key = 1;
  // iv for encrypted data
  bytes iv = 2;
  // encrypted data
  bytes encrypted_data = 3;
}

message EncryptedServerHelloData {
  // signature to authenticate handshake values
  bytes signature = 1;
  // challenge for caller authentication
  bytes challenge = 2;
  // used for hops to tell next hop how to address prev hop
  uint32 backward_frame_id = 3;
  // secret forward frame id
  uint32 forward_frame_id = 5;
}

message RoutingInformation {
  // signal peer if it is the endpoint
  bool is_endpoint = 1;
  // next hop's address, set if peer is not the endpoint
  bytes next_hop_addr = 2;
  uint32 next_hop_port = 3;
  // response, which can be used for caller authentication in ↩
    future work
  oneof optional_challenge_response {
    bytes challenge_response = 4;
  }
  // tunnel_update_reference signals if the new tunnel is a ↩
    rebuilt
  uint32 tunnel_update_reference = 6;
  // signal if this tunnel is only used for cover traffic
  bool cover_only = 7;
}

message ApplicationData {
  // sequence number for replay protection
  fixed32 sequence_number = 1;
  // application payload
  oneof message {
    // data payload
    bytes data = 2;
    // cover payload
    CoverTraffic cover_traffic = 3;
    // close
    Close close = 4;
  }
}

message CoverTraffic {
  // cover traffic
  bytes data = 1;
  // cover traffic must be mirrored by the receiver once
  bool mirrored = 2;
}

message Close {}
```

Remark. *One disadvantage of using protobuf is that a fixed packet size is required while protobuf makes use of dynamic length prefixes. Here, the length descriptions in the serialized data used in protobuf is more of a hindrance than a help, since we had to calculate all the allowed maximal sizes that in the end exactly leads to the 1024-bit packet size. When own header formats would be used, the dynamic length fields could be removed, which increases the number of payload bytes per frame. This is future work.*

2.2.6 Handshake Protocol

This section describes the tunnel handshake protocol in a single-hop scenario. It is summarized in figure 16. Messages between the directly communicating peers will be exchanged using a secure DTLS channel. In the beginning, each peer owns a RSA identity key pair. From the CM or the RPS Module, Alice knows the addresses and the public keys of the Hop and Bob. Further, the public identity keys of each participant are bounded to the current peer's address via the short-lifetime identity certificate, issued by the trusted PKI server. Note that the frame IDs have been removed in this description. Further, the security aspects are analysed in section 3.

Step 1: ClientHello from Alice to Hop
Alice starts with choosing an elliptic curve E and a generator point G. She then selects a private ECDHE parameter a and calculates her public parameter aG. Then she sends the *ClientHello = (E, G, aG)* to Hop. Note that the ClientHello must not contain any integrity or authenticity protection, since Alice must not leak her identity to Hop.

Step 2: ServerHello from Hop to Alice
The Hop receives the ClientHello from Alice and uses the provided curve and generator point to generate its own ECDHE parameter pair *(h, hG)*. It then calculates the shared secret via the ECDHE $s = h \times aG$ and derives two MAC and two ENC keys, one per direction (from Alice to Hop and from Hop to Alice, similar how TLS is doing, to have an additional protection in one direction if the other direction's key is leaked and to avoid IV problems). Since the Hop does not know yet, whether it is an intermediate hop or the target, it acts as it is the target. Therefore, it creates a challenge for a CR (challenge response), which is used for authentication of Alice to the target. The hop then signs his public ECDHE parameter, the challenge and the ClientHello (in exactly this order for replay protection) using its private RSA identity key. Similar to the STS (station-to-station) protocol, the signature is encrypted by the encryption key from Hop to Alice, to avoid that eavesdropper, who know the public key of the Hop, could verify the signature and leak the identity of the Hop. Note that due to the DTLS connections between the peers, an eavesdropper would have to break a DTLS tunnels to access the ServerHello messages. Until the DTLS channel is not compromised, only the previous peer (here Alice, but later other hops) will be able to read the unencrypted ServerHello, thus the encryption of the signature only prevents the previous peer by identifying it. Further note, that we have decided to also encrypt the remaining information of the ServerHello (i.e. challenge and frame IDs), since the keys are already available. Then, the ServerHello is sent back to Alice, containing the public ECDHE parameter and the encrypted part.

Step 3: RoutingInformation from Alice to Hop
Alice first calculates the shared secret and derives the same MAC and ENC keys

as Hop. She then decrypts and verifies the signature to ensure that the data has not been changed (integrity) and the data is actually from Hop (authenticity). Then Alice creates the RoutingInformation, which will tell Hop that it should send the data to the address of Bob. This information is encrypted to ensure nobody else than Hop knows the next hop and it is protected with a MAC to ensure the integrity of the information. Encrypted data and MAC is then sent to Hop.

Step 4: Processing RoutingInformation at Hop
The hop first verifies the MAC and then decrypts the routing information, which tells itself that it is an intermediate hop and all the data should be transmitted to the peer at Bob's address.

Step 5: ClientHello from Alice to Bob
Alice then uses the same generator element and elliptic curve to create a new ECDHE parameter pair *(a2, a2G)*, similar to Step 1. She then encrypts the ClientHello for Bob, using the encryption key from Alice to Hop (or when there are multiple hops, layered encryption). This encrypted ClientHello will then be sent to Hop.

Step 6: Hop delegates data to Bob
Hop then receives the encrypted ClientHello, decrypts it using forward encryption key and sends the data to Bob.

Step 7: ServerHello from Bob to Alice
This step is exactly the same as Step 2, where the Hop is replaced by Bob and the ServerHello is sent back to the Hop (source address of the ClientHello).

Step 8: Hop delegates data to Alice
Hop then receives the plain ServerHello, encrypts it using his backward encryption key and sends the data to Alice.

Step 9: RoutingInformation from Alice to Bob
Alice first decrypts the ServerHello using layered encryption. Since Bob is the target, the challenge of Bob has to be answered by signing the challenge using the private RSA identity key of Alice. This is done for proving authenticity from Alice to Bob. The signature is then packed into the RoutingInformation frame, which is encrypted end-to-end using the encryption key to Bob. Further the MAC is calculated. Then the encrypted data and the MAC is tunneled via the hops. Note that Bob will need the signed certificate of Alice's public identity key for verifying the response of the challenge. However, due to the space limitations (1024-byte packet length) the certificate cannot be sent via the RoutingInformation. A solution to this problem would be to allow fragmentation within the handshake. Since the design of the PKI is future work and also the caller authentication is not supported by the CM yet, handshake fragmentation is also future work and thus no certificate will be sent for now.

Step 10: Hop delegates encrypted RoutingInformation to Bob
Hop then receives the double encrypted RoutingInformation, decrypts it using its forward encryption key with Alice and sends the data to Bob.

Step 11: Processing RoutingInformation at Bob
Bob first verifies the MAC and then decrypts the routing information, which tells himself that he is the target. Then he would verify the solved challenge-response for caller authentication via the provided identity certificate of Alice, which is not available yet and is future work. Afterwards, the connection is established and application data can be exchanged between Alice and Bob.

Remark (Modification in Implementation)**.** *The following changes have been made in the implementation:*

- *Instead of using ENC and MAC keys, the module uses 128-bit AES-GCM, therefore there is only one symmetric session key per direction. The used KDF is simply the 256-bit SHA256 hash of the shared secret, which is split into the two 128-bit keys.*
- *Elliptic curve is always SECP256K1, therefore only the public ECDHE parameters are exchanged during the handshake.*
- *Instead of calculating the ECDHE parameters by our own, the Onion Module makes use of the the openssl interface, which uses elliptic curve key pairs for deriving shared secrets.*

2.2.7 Message Codec

For the procedure of sending and parsing the different onion tunnel messages, a lot of sub-tasks, such as encryption and integrity checks, serialization, data fragmentation, sequence number checks and message forwarding as intermediate hop, need to be done. For simplification, the *MessageCodec* takes care of all these steps. A distinction is made into three different types of codecs, one for the tunnel initiator, one for the tunnel target and one for intermediate hops. For encryption, a so-called *CryptoContext* is provided, which is used for the encryption for one layer between two peers. It holds the two encryption keys, one per direction, manages the freshness-checks and the selection of fresh IVs and provides an interface for encryption/decryption via AES-CTR and authenticated encryption using AES-GCM. An intermediate hop always makes use of AES-CTR mode, while the target codec only uses AES-GCM for end-to-end integrity protection. The initiator codec does not holds only one context but one per encryption layer. After each successful key-exchange the FSM updates the codec and adds the next *CryptoContext* to the codec. When sending a message, the initiator then iterates over all contexts for encryption, using AES-GCM for the first and AES-CTR for all the others.
Further, the target and the initiator codecs own a *SequenceNumberContext*,

which is responsible for providing consecutive outgoing sequence numbers for application data and for checking the freshness of incoming sequence numbers.

2.2.8 DTLS Secure Channels

In addition to the secure tunnels, the Onion Module ensures that each two directly communicating peers exchange messages via a secure DTLS channel, similar to the TOR network. Here, only one DTLS channel between two peers exists at a time and all tunnels, for which these two peers are consecutive, are going through this channel. This is done for authenticity, availability and privacy reasons.
Reaching mutual authentication, which is required to avoid Man-In-The-Middle attacks, is a challenge as a new root of trust is required. Imagine there is a tunnel from Alice via Bob to Charlie, so Alice has to know Bob from RPS and Charlie either from the CM or from RPS as well. Then DTLS connections between Alice and Bob and between Bob and Charlie are necessary. However, Alice cannot be sure that Bob and Charlie know each other and further Alice must not authenticate herself to Bob via the tunnel establishment, such that Bob does not know that Alice is the caller. Therefore, also Bob might not know Alice yet. Since there is no possibility in the lower level modules to request a public key belonging to a specific peer address, there must be any additional root of trust for this purpose. The construction of this root of trust is future work. For simplification we assume that there is some kind of trusted PKI server that provides a self-signed root certificate, used for verifying certificates issued by the PKI server. For now we do not have any kind of hostnames, but use ip addresses and ports for addressing peers. Thus the peer certificates bind the public identity keys to the peer's address, which is the SHA256 hash of *"$< ip >$::$< port >$"*. The infrastructure is shown in figure 15.

The project specification requires that connections to peers that misbehave must be terminated immediately. After disconnecting from a peer, this peer is then blocked for a configurable duration. For availability reasons, a distinction should be made between misbehaviour that is triggered by the previous peer and misbehaviour that can be forced by other peers. Otherwise, outsider attacker peers could exploit this for denial of service attacks. Attackers could for example modify or block messages of the DTLS channel. As long as the channel is not established, this leads to a handshake failure. In the connected channel, invalid packets will be discarded by the receiving peer, since the integrity checks will fail, while blocking messages could for example lead to tunnel handshake failures. Further, an attacker could be the initiator of a new tunnel, while the tunnel handshake protocol procedure is not followed or integrity checks of tunnel messages fail. All the mentioned misbehaviour so far, can be triggered by other peers and must therefore not result in disconnecting from the whole peer. On the other hand, when the previous peer receives a package that should be transferred, it has to check the packet size, whether the received data can be parsed into a tunnel frame message and if the IV is fresh. If one of these checks fail,

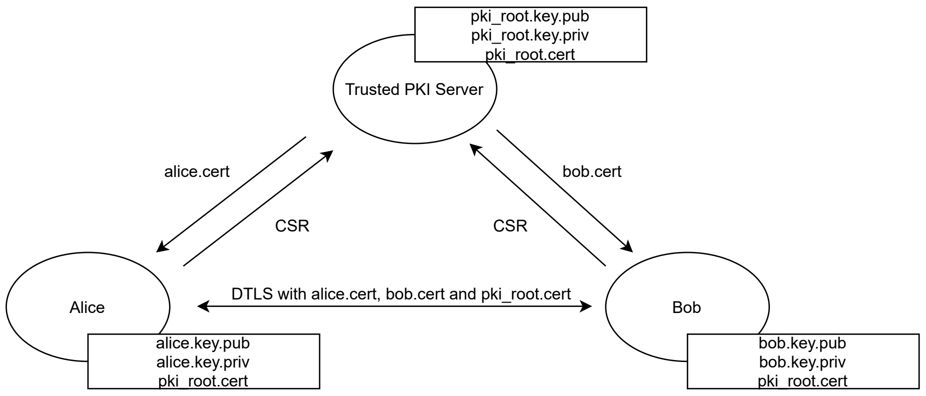

Figure 15: Public Key Infrastructure for DTLS authentication: A trusted PKI server issues certificates, requested via any kind of Certificate Signing Request (CSR) to the peers. The root certificate is used as root of trust.

the previous peer must not forward the package. Thus, when our peer receives such an ill-formed packet, this misbehaviour is fully triggered by the previous peer and we have to disconnect from it immediately. Another question is how we should handle incoming packets with an invalid frame ID. Since the frame cannot be mapped to a corresponding tunnel, only the whole connection to the previous peer could be terminated. However, there might be the case that the receiving peer has already terminated the tunnel with the corresponding frame ID, due to e.g. a new round, while the previous peer still forwards some packets from the caller. In this case it is not a misbehaviour but a non-avoidable race condition, which is why the Onion Module simply ignores packets with invalid frame IDs.

The DTLS channels also improve privacy, since the frame IDs are sent encrypted. If this were not the case, global traffic observers could count the number of packets per frame ID, which might be a unique fingerprint and could therefore reveal full tunnel routes.

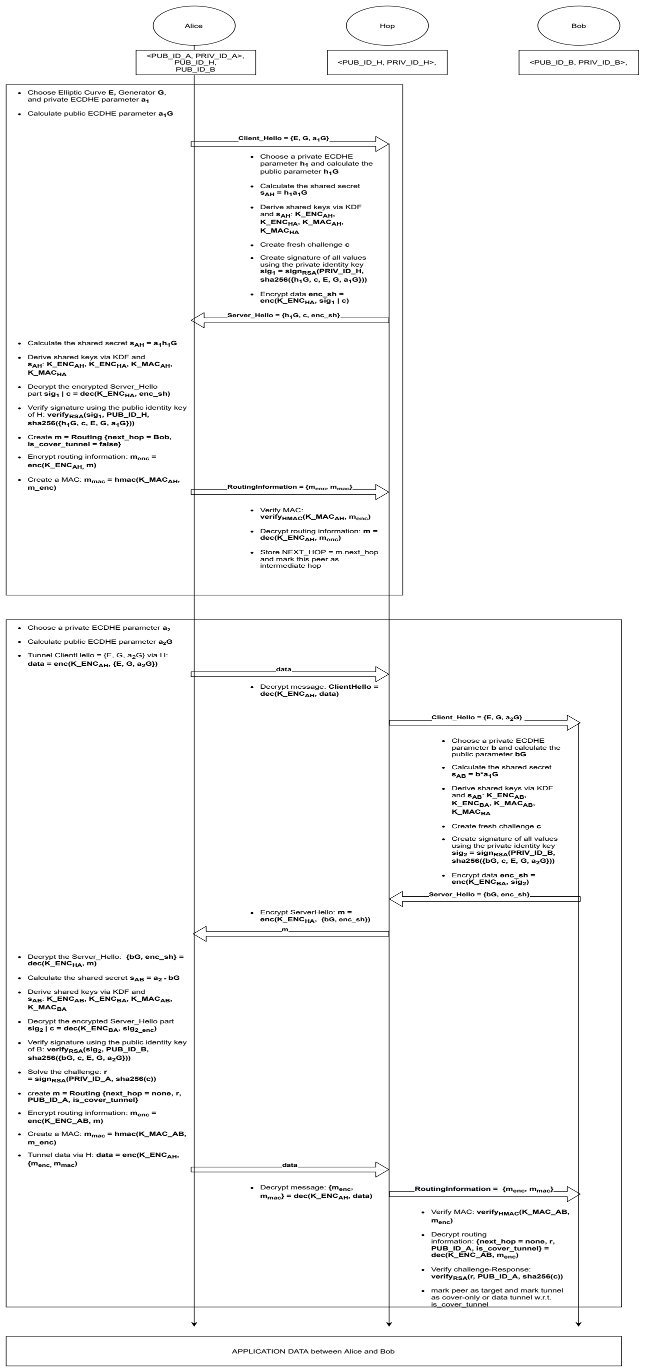
Alice
Hop
Bob
<PUB_ID_A, PRIV_ID_A>, PUB_ID_H, PUB_ID_B
<PUB_ID_H, PRIV_ID_H>,
<PUB_ID_B, PRIV_ID_B>,
Choose Elliptic Curve **E,** Generator **G**, and private ECDHE parameter $\mathbf{a_1}$
Calculate public ECDHE parameter $\mathbf{a_1G}$
Client_Hello = $\{E, G, a_1G\}$
Choose a private ECDHE parameter $\mathbf{h_1}$ and calculate the public parameter $\mathbf{h_1G}$
Calculate the shared secret $s_{AH} = h_1a_1G$
Derive shared keys via KDF and s_{AH}: K_ENC_{AH}, K_ENC_{HA}, K_MAC_{AH}, K_MAC_{HA}
Create fresh challenge **c**
Create signature of all values using the private identity key $sig_1 = sign_{RSA}(PRIV\_ID\_H, sha256(\{h_1G, c, E, G, a_1G\}))$
Encrypt data **enc_sh =** $enc(K\_ENC_{HA}, sig_1 \mid c)$
Server_Hello = $\{h_1G, c, enc\_sh\}$
Calculate the shared secret $s_{AH} = a_1h_1G$
Derive shared keys via KDF and s_{AH}: K_ENC_{AH}, K_ENC_{HA}, K_MAC_{AH}, K_MAC_{HA}
Decrypt the encrypted Server_Hello part $sig_1 \mid c = dec(K\_ENC_{HA}, enc\_sh)$
Verify signature using the public identity key of H: $verify_{RSA}(sig_1, PUB\_ID\_H, sha256(\{h_1G, c, E, G, a_1G\}))$
Create **m = Routing {next_hop = Bob, is_cover_tunnel = false}**
Encrypt routing information: $m_{enc} = enc(K\_ENC_{AH}, m)$
Create a MAC: $m_{mac} = hmac(K\_MAC_{AH}, m\_enc)$
RoutingInformation = $\{m_{enc}, m_{mac}\}$
Verify MAC: $verify_{HMAC}(K\_MAC_{AH}, m_{enc})$
Decrypt routing information: $m = dec(K\_ENC_{AH}, m_{enc})$
Store NEXT_HOP = m.next_hop and mark this peer as intermediate hop
Choose a private ECDHE parameter $\mathbf{a_2}$
Calculate public ECDHE parameter $\mathbf{a_2G}$
Tunnel ClientHello = $\{E, G, a_2G\}$ via H: $data = enc(K\_ENC_{AH}, \{E, G, a_2G\})$
data
Decrypt message: **ClientHello =** $dec(K\_ENC_{AH}, data)$
Client_Hello = $\{E, G, a_2G\}$
Choose a private ECDHE parameter **b** and calculate the public parameter **bG**
Calculate the shared secret $s_{AB} = b{*}a_1G$
Derive shared keys via KDF and s_{AB}: K_ENC_{AB}, K_ENC_{BA}, K_MAC_{AB}, K_MAC_{BA}
Create fresh challenge **c**
Create signature of all values using the private identity key $sig_2 = sign_{RSA}(PRIV\_ID\_B, sha256(\{bG, c, E, G, a_2G\}))$
Encrypt data **enc_sh =** $enc(K\_ENC_{BA}, sig_2)$
Server_Hello = {bG, enc_sh}
Encrypt ServerHello: $m = enc(K\_ENC_{HA}, \{bG, enc\_sh\})$
m
Decrypt the Server_Hello: $\{bG, enc\_sh\} = dec(K\_ENC_{HA}, m)$
Calculate the shared secret $s_{AB} = a_2 \cdot bG$
Derive shared keys via KDF and s_{AB}: K_ENC_{AB}, K_ENC_{BA}, K_MAC_{AB}, K_MAC_{BA}
Decrypt the encrypted Server_Hello part $sig_2 \mid c = dec(K\_ENC_{BA}, sig_{2\_enc})$
Verify signature using the public identity key of B: $verify_{RSA}(sig_2, PUB\_ID\_B, sha256(\{bG, c, E, G, a_2G\}))$
Solve the challenge: $r = sign_{RSA}(PRIV\_ID\_A, sha256(c))$
create **m = Routing {next_hop = none, r, PUB_ID_A, is_cover_tunnel}**
Encrypt routing information: $m_{enc} = enc(K\_ENC\_AB, m)$
Create a MAC: $m_{mac} = hmac(K\_MAC\_AB, m\_enc)$
Tunnel data via H: $data = enc(K\_ENC_{AH}, \{m_{enc}, m_{mac}\})$
data
Decrypt message: $\{m_{enc}, m_{mac}\} = dec(K\_ENC_{AH}, data)$
RoutingInformation = $\{m_{enc}, m_{mac}\}$
Verify MAC: $verify_{HMAC}(K\_MAC\_AB, m_{enc})$
Decrypt routing information: $\{next\_hop = none, r, PUB\_ID\_A, is\_cover\_tunnel\} = dec(K\_ENC\_AB, m_{enc})$
Verify challenge-Response: $verify_{RSA}(r, PUB\_ID\_A, sha256(c))$
mark peer as target and mark tunnel as cover-only or data tunnel w.r.t. is_cover_tunnel
APPLICATION DATA between Alice and Bob

Figure 16: Handshake protocol between the caller Alice and the callee Bob via one intermediate hop

3 Security Analysis

In this section is discussed how the Onion Module tries to reach the individual security goals.

3.1 Authenticity

To avoid e.g. Man-In-The-Middle attacks we have to ensure that the origin of a received message is always checked against the identity of the expected sender. This is done via a signature based on the identity key pair. The Onion Module gets the public identity keys of hops from the RPS Module and the public key of the callee from the CM. Here, it must be ensured that the caller does not authenticate itself to intermediate hops, since the hops must not know the caller's identity for privacy reasons. Thus, the handshake between the caller and intermediate hops is not mutual but single authenticated. Unfortunately, the project environment does not provide any possibility to pass the identity of the caller to the callee's CM, like a phone number that is shown to the callee. This is why any kind of caller authentication at the target peer does not make sense, the callee is simply not interested in knowing the identity of the caller. With the hope that this interface will be upgraded at some point, we have designed caller authentication in the Onion Module via a challenge-response protocol anyway. However, this challenge-response protocol is not used yet.
In the handshake protocol (2.2.6), the callee/intermediate hop signs the public ECDHE parameters, among others, with its private identity key. The caller is then able to verify the signature with the trusted public identity key of the participant peer and can then be sure that the received public ECDHE parameter has its origin at the other peer. Then both, the caller and the other peer can calculate the same secret via the ECDHE, which then leads to secret symmetric keys, which can only be known by the caller and the participant peer due to the properties of ECDHE. Using these keys for encryption and integrity protection ensures authenticity of all exchanged messages.

Remark. *Since we do not use caller authentication yet, there is no full protection against authenticity attacks, since another peer can masquerade as the caller and replace the callers public ECDHE parameter in the ClientHello by its own. However, this is nothing else then blocking the original call and creating an individual tunnel from the attacker node to the callee, since the callee is not interested in the origin of the call.*

In addition to the authenticity of the individual tunnels, the secure channels (DTLS channels) ensure mutual authenticity between each two directly communicating peers (i.e. between the caller and the first hop, between two hops and between the last hop and the callee). This mutual authentication is reached by having a trusted public-key-infrastructure that signs the identity certificates of the peers.

3.2 Availability

To ensure availability of a running peer and of active tunnels could be quite a challenging task and does highly depended on the attacker model used. For example, an attacker who controls a part of a network used for forwarding data packets (e.g. control of wire, router or peer) within a connection can always block data frames. Also a rootkit, installed on the device running the Onion Module, would always be able to shutdown the peer. Therefore the design of the Onion Module is focusing on availability protection against less powerful attackers. It is assumed, that an attacker is able to run multiple malicious peers and can send new DTLS messages between two directly communicating peers. However, since an attacker does not know the key material between two non-controlled peers, inserted DTLS traffic is simply ignored at the peers due to integrity failures. Thus, an attacker cannot manipulate or add tunnel traffic unless it is routed via a peer controlled by the attacker. As discussed in section 2.2.8, tunnel message integrity failures or blocked tunnel frames that might lead to tunnel handshake timeouts can be triggered by other peers (intermediate hops or the initiator peer) and will therefore not result in disconnecting from the peer, but only in closing the tunnel. Thus, if an attacker peer is part of a tunnel route, a denial of service attack on this tunnel can no longer be prevented. Here, it is already necessary to make it more difficult for attackers to participate in the network by means of authenticity checks. However, misbehavior such as sending invalid tunnel frames, reused IVs, or invalid packet length that cannot be produced by third party participants are a clear indication that the previous peer in a tunnel is an attacker. Summarized, an attacker Eve can only shutdown tunnels or DTLS connections when she is directly involved as a peer, can break a DTLS channel (which is not assumed due to the strength of DTLS when configured correctly) or controls networks of other peers.

3.3 Confidentiality

Secret information, such as the application data, routing information and the padding length must be encrypted using the symmetric encryption keys, derived from the shared ECDHE secret. The Onion Module makes use of the layered onion encryption mechanisms, where each hop decrypts forward messages and encrypts backward messages with its symmetric keys. For this, two encryption keys are used, one per direction, to prevent problems with the IV selection. For performance reasons, the AES counter mode (AES-CTR for hops, AES-GCM for the target) with 128 bit keys is used.
One challenge is the selection of the 16-bit initialization vector (IV). As already discussed in 2.2.3, there must not be any additional information for routing or encryption for the hops, due to the fixed packet length. Further, there cannot be a unique trackable IV per packet, that is used for the encryption at every hop. Therefore, the Onion Module takes the 16-bit IV from the packet and encrypts it with the symmetric key using AES-ECB. Due to the injectivity of AES and the uniqueness of the IV selected at the caller or the callee, this procedure produces

unique, non-trackable IVs for every hop. Using AES-ECB is fine here, since the 16-bit IV is a fresh single block. Once a peer received a frame containing a reused IV, the tunnel must be closed to prevent tracking attacks. The procedure is shown in figure 17 for single-hop forward messaging. Furthermore, the DTLS

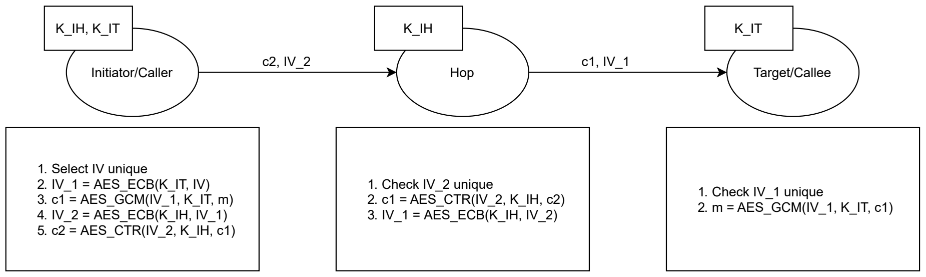
K_IH, K_IT
Initiator/Caller
c2, IV_2
K_IH
Hop
c1, IV_1
K_IT
Target/Callee
1. Select IV unique
2. IV_1 = AES_ECB(K_IT, IV)
3. c1 = AES_GCM(IV_1, K_IT, m)
4. IV_2 = AES_ECB(K_IH, IV_1)
5. c2 = AES_CTR(IV_2, K_IH, c1)
1. Check IV_2 unique
2. c1 = AES_CTR(IV_2, K_IH, c2)
3. IV_1 = AES_ECB(K_IH, IV_2)
1. Check IV_1 unique
2. m = AES_GCM(IV_1, K_IT, c1)

Figure 17: Single-hop forward messaging using layered encryption

channels between two directly communicating peers ensure the encryption of the individual tunnel frames and their frame IDs.

3.4 Integrity

Data-integrity is the goal of protecting data from unauthorized modification. Starting with the last handshake message (*RoutingInformation*), all further messages are encrypted end-to-end using the AES-GCM mode, which ensures data-integrity, assumed the shared ECDHE secret is trusted and the key-derivation function is safe. The integrity of the second handshake message (*ServerHello*) is ensured by the signature over the ECDHE parameters, assumed that the public identity key of the peer is trusted, which is the case. As already described in 3.1, hops must not know the caller's identity, while the callee is not interested in knowing the caller's identity. However, the caller chooses its ECDHE public and private parameters fresh. Thus, no attacker is able to know the caller's private ECDHE parameter due to the strength of elliptic curve cryptography. Since the public ECDHE parameter of the caller is one of the signed values in the *ServerHello*, the caller can be sure that the other peer knows the caller's correct public ECDHE parameter for calculating the shared secret. Summarized, the caller can trust the ECDHE secret when the signature is verified, while the other peer is not interested in knowing the caller's identity and therefore also trusts the shared secret.

3.5 Privacy

Given a tunnel from the caller Alice over multiple hops (at least two) to the callee Bob, the following privacy requirements must hold:

1. Tunnel establishment must be transferred via the previous partial-established tunnel, such that there is no direct communication between Alice and Bob, which could be detected by a local observer or any network sniffer.

2. An intermediate hop must not know its position within the tunnel route. Thus, an intermediate hop must only know its direct successor's and predecessor's address, but not their functions.

3. Only Alice and Bob are allowed to know that they are communicating with each other.

4. No trackable information within a data frame must be sent. This means that all information of an incoming data frame (Frame ID, IV, data) must be fully replaced in the outgoing frame at an intermediate hop, such that there is no observable correlation between the incoming and the outgoing information.

5. No privacy-leaking information must be send within a data frame or within the handshake.

The following enumeration discusses how the privacy requirements are achieved:

1. This is how the protocol is designed.

2. Let us first assume that there are no timing attacks. Each intermediate hop obtains the address of the previous hop through the source address of the packets and the address of the next hop through the end-to-end encrypted routing information. Due to the encrypted DTLS connections between peers and the large amount of anonymity (due to the cover traffic), a per cannot easily observe the incoming traffic of the previous peer and associate it with the actual tunnel traffic, and therefore is not able to find out if the previous peer is the initiator of the call. This is also true for the outgoing traffic of the next hop. In addition, Alice never authenticates herself to an intermediate hop, so the identity of the caller is completely hidden to an intermediate hop. Since the next hop's function (destination or intermediate hop) is sent end-to-end encrypted, the intermediate hop cannot know whether the next hop is Bob or another intermediate hop. Bob authenticates to Alice via the end-to-end encrypted signature in the ServerHello, so Bob's identity is also not known to any of the intermediate hops. Since no other addresses or identities are exchanged in the protocol, an intermediate peer can say nothing about its position within the route and thus nothing about the function of the previous or next hop. However, there is the problem of passive and active timing attacks, since the rounds are synchronized and the handshake of the first hop must occur before the handshake of the second hop, etc. In addition, network attackers capable of blocking outgoing packets from other participants could delay frames. If such a delayed frame is a handshake message for a peer controlled by the attacker, then the attacker knows an earlier hop of the tunnel route. Preventing such timing attacks is a challenge. The handshake timeout in the

Onion Module already prevents excessive delays. Further countermeasures will be future work.

3. This follows from the previous statement. Since no intermediate hop know its position within the route, no hop knows who is the caller and the callee.

4. We assume that the current peer is not controlled by an attacker. Then the frame ID is replaced by the frame ID for addressing the next peer, the IV is encrypted using AES-ECB, combined with the session key and the data are encrypted using AES-CTR (or AES-GCM). Thus all component of the incoming frame are replaced by different values in the outgoing frame. Further, tracking based on counting packets per frame IDs as a global observer is not possible due to the DTLS channels between the peers, which encrypts the frame IDs.

5. Privacy-leaking information are information that map to the identity (public identity key or address) of another peer. Obliviously, Alice is allowed to know all the identities of the selected peers. An intermediate hop only sees the ClientHello and the unencrypted part of the ServerHello between the caller and the next hop, as well as the local frame IDs and the IVs in the incoming packets. The IVs are fresh and do not correlate to the identity of peers when not knowing the session keys of the other peers. The ECDHE parameters and the local frame IDs do also not map to any identity of another participant than the next and the previous hops. The target peer only knows the identity of the caller via the challenge-response protocol for caller authentication and the address of the previous hop.

3.6 Perfect Forward Secrecy (PFS)

Assuming that the caller Alice and the peer Bob use any kind of non-ephemeral Diffie Hellman, then every time when Alice and Bob are doing a key-exchange, the resulting shared secret would be the same. Once a shared secret or a symmetric encryption key is leaked, the previous recorded communication could be decrypted completely. Further, the exchange of the public Diffie Hellman parameters is nothing else then exchanging the identities of the peers, what violates privacy. Therefore, ephemeral (elliptic curve) Diffie Hellman (only ECDHE) is used in the Onion Module.

3.7 Replay Protection

Replay attacks try to compromise the authenticity of data by repeating recorded valid transmissions at a later time. Therefore, it has to be ensured that every packet contains any kind of freshness (nonce or timestamp) that is checked when receiving the frame. In the Onion Module, the *ClientHello* and *ServerHello* contain fresh ECDHE parameters. The *RoutingInformation* message is fresh, since it is encrypted by the fresh encryption key and is only expected once, similar to the message that passes the *Forward Frame ID* to the next hop. The application data frames contain a sequence number that is unique in combination with the temporary encryption key of the tunnel. Therefore, all tunnel messages contain freshness and thus replay protection.

3.8 Fixed Packet Size

To avoid packet-size analysis attacks, the Onion Module ensures that each frame has the same amount of bytes (1024 bytes). This is done by adding secret padding to the payload data.

4 Software Documentation

4.1 License

This project is licensed under the Apache License 2.0.

4.2 Operating Systems

Tested on macOS Big Sur 11.2.1, Archlinux 5.13.13 and Debian Buster 5.14.0.

4.3 Dependencies

1. **Rust Toolchain**: latest stable release can be installed via https://rustup.rs/ (tested on version 1.54.0)

2. **Openssl** (tested on versions >= 1.1.1d)

3. **Protobuf Compiler**: On Linux via *apt-get install protobuf-compiler*, on macOS via *brew install protobuf* (tested on versions >= 3.6.1)

4.4 Building, Testing and Running

The application can be built, tested and run via the *cargo* package manager:

```
// Build: Run in the root directory
cargo build

// Run: Run in the root directory
cargo run -- -c <path-to-config-file>

// Run with help menu
cargo run -- --help

// Test: Run in the root directory
cargo test

// Set log level via RUST_LOG environment VAR
RUST_LOG=trace cargo run -- -c <path-to-config-file>
RUST_LOG=trace cargo test
```

4.5 Configuration File

The binary expects a path to an INI configuration file, which must contain the following parameters:

Attribute	Section	Type	Description
hostkey	global	str	Path to the peer's public hostkey in PEM format
private_hostkey	onion	str	Path to the peer's private hostkey in PEM format
hostkey_cert	onion	str	Path to the peer's certificate in PEM format
pki_root_cert	onion	str	Path to the PKI root certificate in PEM format
p2p_port	onion	u16	P2P port of peer
p2p_hostname	onion	str (hostname, ipv4, ipv6)	P2P hostname/IP
hop_count	onion	u8	Number of intermediate hops per tunnel (>= 2)
api_address	onion	str (hostname:port, ipv4:port, [ipv6]:port)	Peer's API address for the onion layer
round_time	onion	u64 (seconds)	Round time (default=600s)
build_window	onion	u64 (ms)	Build window (default=1000ms)
handshake_timeout	onion	u64 (ms)	Handshake message timeout (default=1000ms)
timeout	onion	u64 (seconds)	Timeout to recognize inactive tunnels (default=15s)
blocklist_time	onion	u64 (seconds)	Seconds until a peer is blocked after misbehaviour (default=3600s)
api_address	rps	str (hostname:port, ipv4:port, [ipv6]:port)	Peer's API address for the RPS layer

Here, the PKI is the root of trust, that issues certificates for the peers. The *pki_root_cert* is the self-signed root certificate which contains the public key of the PKI, which is used for verifying the signature of the *hostkey_cert*. This public key infrastructure is used for the DTLS connections between the peers (see 2.2.8).

A example for an INI config file can be found in the root directory of the project, called *template.config*. All parameters without default value are mandatory.

5 Future Work

1. Protection against passive and active timing attacks.
2. Design of a secure PKI with trusted PKI server for issuing short-lifetime certificates.
3. Handshake Fragmentation to allow the sending of the signed identity certificate from the caller to the callee for caller authentication.
4. Caller authentication: Pass the identity of the caller to the callee's CM.
5. Replace protobuf message formats by own header format to decrease length descriptions and increase throughput.

6 Workload Distribution and Effort

- Leon Beckmann (Effort: 200h)
  - Implementation of the API protocol
  - Implementation of the ConfigParser
  - Implementation of the RPS API
  - Design and Implementation of the FSM inclusive handshake FSM
  - Implementation of the onion tunnels
  - Improvements in the p2p_protocol package
  - Message format design and implementation of padding and fragmentation
  - Handshake protocol design (together)
  - Implementation of the integration test
  - Design and implementation of the round synchronizer
  - Implementation of cover tunnels and cover traffic
  - Implementation of multiple FrameIDs (has been removed again)
  - Project Documentation (Finale Report)
- Florian Freund (Effort: 140h)
  - Setting up Gitlab CI
  - First draft of the p2p_protocol implementation
  - Dynamic tunnel identifier protocol (has been removed again)
  - Handshake protocol design (together)
  - Implementation of the crypto stuff, inclusive handshake
  - Unit tests
  - Small code improvements
  - Implementation of DTLS channels and the connection blocklist

References

[1] Roger Dingledine and Nick Mathewson. *Tor Protocol Specification*. `https://gitweb.torproject.org/torspec.git/tree/tor-spec.txt`. [Online; accessed 11-September-2021]. 2021.

[2] Sree Harsha Totakura et al. *VoidPhone Project Specification*. June 2021.