Voidphone Project: Midterm Report

Oliver Jacobsen (03696596), Dominik Stolz (03696456)

July 12, 2020

Introduction

Since the phase of initial planning, quite some progress was made towards the actual implementation. As of now, both the API and the P2P protocol are implemented, as well as the basic architecture of the program. The main missing parts are the seamless tunnel switch-over mechanism and graceful deconstruction of tunnels. During the design, it became apparent that some of the assumptions made in the initial report ought to be revised. Instead of using the `async-std` crate as async IO runtime, we opted for the `tokio` crate which is more mature and provides more features. We initially chose `async-std` for its better performance and similarity to the standard library. However, it turned out to be inferior to `tokio` in regards to practicability in network applications.

Further, we chose to implement the parsing and the serialization of packets by hand as opposed to using a library such as `protobuf` for that purpose. This gave us more flexibility in the design of our protocol – especially considering the requirements imposed by the encryption-wrapping nature of the onion protocol.

Architecture

Logical Structure

Cargo – the default build system for Rust – allows to include a library as well as multiple binary artifacts in a single crate. So, we split our application into a binary handling API connections and a library servicing requests from these connections. This enables the possibility to reuse all of the onion functionality outside the context of the Voidphone project. For this reason, we tried to keep the public interface of the library as general as possible and confine all Voidphone specifics to the binary. This separation also reflects in our directory structure:

```
src
├── api
│   ├── config.rs
│   ├── mod.rs
│   ├── protocol.rs
│   ├── rps.rs
│   └── socket.rs
├── lib.rs
├── main.rs
├── onion
│   ├── circuit.rs
│   ├── crypto.rs
│   ├── mod.rs
│   ├── protocol.rs
│   ├── socket.rs
│   ├── tests.rs
│   └── tunnel.rs
└── utils.rs
```

The file `main.rs` contains the entry point for the binary and uses modules in the `api` directory. It also uses the included library, for which `lib.rs` is the root module. The library uses modules from the `onion` directory. The module `utils.rs` is used by both the binary and the library.

The library is in turn designed in multiple layers – the *P2P Protocol* constituting the bottom most one and the *Public Interface* the top most one. In between there is the *Circuit Layer* which is mostly concerned with relaying messages from one hop to the next. On top of the *Circuit Layer* resides the *Tunnel Layer* which represents the end-to-end view. The table in figure 1 shows an overview of all logical layers with the corresponding source files. The *Onion Socket* layer serves as glue between higher-level intentions, such as initiating a handshake, and actually writing serialized packets to a socket. The purpose of the *Round Handler* is to track all outgoing and incoming tunnels and schedule operations requested by the *Public Interface*.

Layer	Source File
Public Interface	`lib.rs`
Round Handler	`lib.rs`
Tunnel Layer	`onion/tunnel.rs`
Circuit Layer	`onion/circuit.rs`
Onion Socket	`onion/socket.rs`
P2P Protocol	`onion/protocol.rs`

Figure 1: Overview of layers and source files

Circuit Layer

A circuit is defined as the direct connection between two peers and is part of a tunnel involving multiple hops. In the code it is represented with the `Circuit` struct (cf. figure 2) which stores an unique identifier and a socket for communicating with the other peer.

```
struct Circuit {
    id: CircuitId,
    socket: Mutex<OnionSocket<TcpStream>>,
}
```

Figure 2: Circuit struct

For each incoming connection, a key exchange is performed first before handling arriving messages. Here it is important to note that keys are not necessarily exchanged with the connecting peer, but with the peer who is building the tunnel. After a successful key exchange, a `CircuitHandler` struct (cf. figure 3) is created, which also stores the negotiated session key.

```
struct CircuitHandler {
    in_circuit: Circuit,
    session_key: SessionKey,
    state: CircuitHandlerState,
    // omitted other fields for brevity
}
```

Figure 3: CircuitHandler struct

The `CircuitHandler` can be in one of three states:

```
enum CircuitHandlerState {
    Default,
    Router {
        out_circuit: Circuit,
    },
    Endpoint {
        tunnel_id: TunnelId,
        requests: Receiver<Request>,
    },
}
```

Figure 4: CircuitHandlerState enum

In the `Default` state, the handler just waits for a message instructing it to either extend the tunnel and become a `Router` or to be the `Endpoint` of a tunnel. In the `Router` state, the handler forwards messages in both directions until the tunnel is truncated or destroyed. In the `Endpoint` state, the handler connects its `in_circuit` to the *Tunnel Layer*.

Tunnel Layer

The *Tunnel Layer* is further subdivided into outgoing and incoming tunnels. In code, incoming tunnels exist implicitly as `CircuitHandler` in the `Endpoint` state. Outgoing tunnels are represented through the `Tunnel` struct (cf. figure 5), which stores the circuit to the first hop in the tunnel as well as a list of the session keys negotiated with each hop.

```
struct Tunnel {
    id: TunnelId,
    out_circuit: Circuit,
    session_keys: Vec<SessionKey>,
}
```

Figure 5: Tunnel struct

Process Architecture

Tasks and Channels

The process architecture of our application is based on a single asynchronous Rust program spawning various concurrent tasks. These tasks are executed on an event loop provided by the `tokio` crate, which internally uses a thread pool for scheduling. The main tasks are:

- For each incoming API connection a handler task is spawned, which calls the corresponding methods in the library.
- The library spawns a `RoundHandler` task which periodically builds and destroys tunnels.
- Similarly to the API handler tasks, a `CircuitHandler` task is spawned for every incoming connection on the P2P port.
- A task is spawned for each created tunnel, to handle messages sent back over the tunnel.

Communication between tasks is done in a message passing fashion using channels. For instance, calling the `build_tunnel` method in the library sends a `Build` request over a channel which is then consumed by the `RoundHandler`. Synchronization using message passing often allows to avoid shared state, which is a common source of race conditions and related bugs.

To notify the API on incoming tunnels or received data, another channel is used, which propagates events up from the circuit and tunnel handlers. An overview of the communication schema is shown in figure 6.

Networking

Networking is also done in an asynchronous manner using the methods provided by `tokio`. A call to `read` creates a future, which can be awaited inside of a coroutine. The `tokio` runtime then uses `epoll` to resume the coroutine once the I/O operation completed.

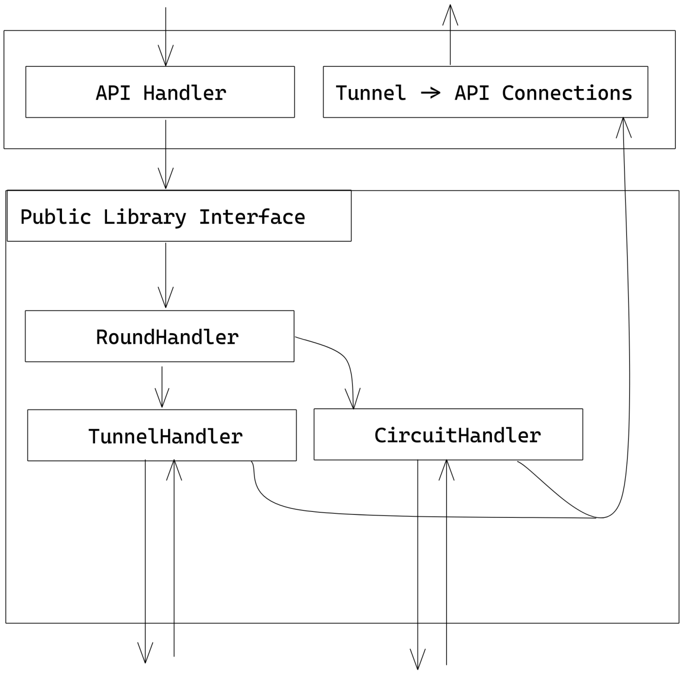


Figure 6: Concurrent tasks and their communication channels

Peer-to-Peer Protocol

Our peer-to-peer protocol has been highly inspired by the Tor protocol. We differentiate between circuit-level packets, which are targeted at the directly connected peer as part of a tunnel, and tunnel-level packets, which enable encrypted communication across several peers. In the following, the first peer in a tunnel is called tunnel controller (TC), the final peer is the destination (D), and all other peers are hops (H). An overview of the tunnel construction and deconstruction is illustrated in figure 19.

Circuit-Level Packets

The circuit-level packets are Create, Created, Opaque and Teardown. Create and Created are part of the circuit handshake, Opaque functions as a shell for tunnel-level packets and Truncate is used to terminate circuit connections between peers. Each circuit packet contains a circuit ID, which is used to identify which circuit a packet belongs to. This enables peers to multiplex many circuits onto a single TCP connection.

All packets are padded to have a fixed length of 1024 bytes.

The Circuit-Handshake

The circuit handshake is used to setup a new circuit and agree on a session key with the TC for the encryption of tunnel messages. In order to initiate a new circuit, a peer sends a Create (cf. figure 7) message to another peer. As payload, the Create message contains an ephemeral public key of the TC.

If accepted, the targeted peer responds with a Created (cf. figure 8) message containing its ephemeral public key for the key exchange. In order to prevent any other peer than the requested peer from responding to a Create message, the key in the Created message is signed using RSA. The receiver can therefore verify the signature with the corresponding public key, which was previously acquired out-of-band.

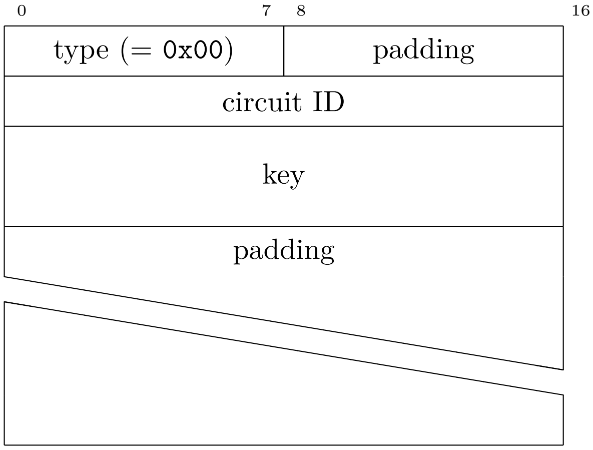


Figure 7: CircuitCreate

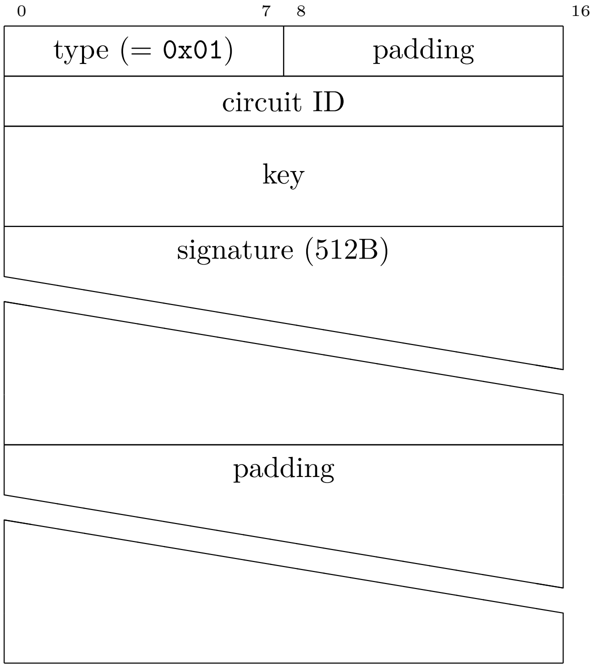


Figure 8: CircuitCreated

Circuit Opaque Messages

Once the circuit handshake is complete, tunnel messages can be transmitted via a circuit using Opaque packets. The encrypted tunnel message is transmitted as the payload of an Opaque (cf. figure 9) message. In addition to the payload, the packet also contains the SHA-256 digest of the payload as a checksum.

The receiver decrypts the digest and payload using the session key generated during the circuit handshake and the nonce provided by the Opaque header. Next, the receiver computes a SHA-256 digest of the decrypted payload. If the digest sent in the packet matches the computed digest, the tunnel message is interpreted by the peer. Else, the peer is expected to forward the decrypted received digest and payload to the next hop as the payload of an Opaque message.

We chose to use a unique nonce in the Opaque message in order to improve the cryptographic strength of the AES encryption. This way, a hop close to the endpoint of a tunnel will not be able to use pattern recognition in order to leak information from the payload of a stream of Opaque messages.

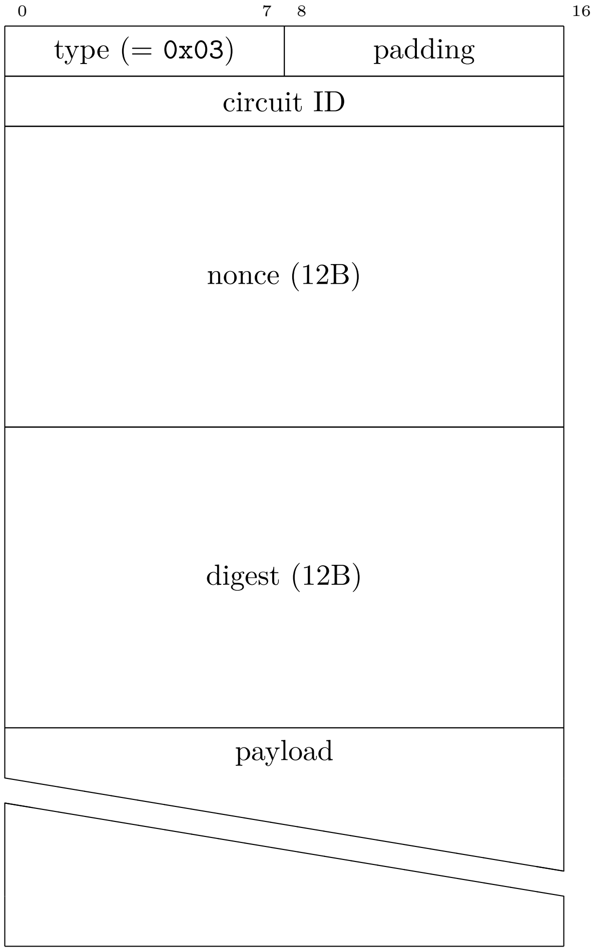


Figure 9: CircuitOpaque

Circuit Teardown

The Teardown (cf. figure 10) packet can be used to signal a peer that the connection has been reset. Once a Teardown message is sent, a peer is no longer required to keep a circuit alive or listen for incoming packets. Teardown messages may be sent at any time via a circuit and it is the responsibility of the receiving peer to react and propagate any errors. However, there is also no guarantee that a peer sends a Teardown message before breaking down the connection.

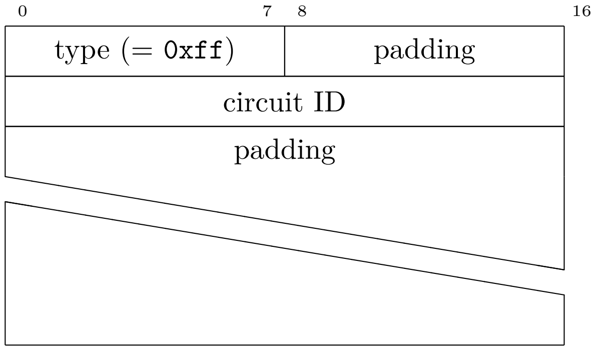


Figure 10: CircuitTeardown

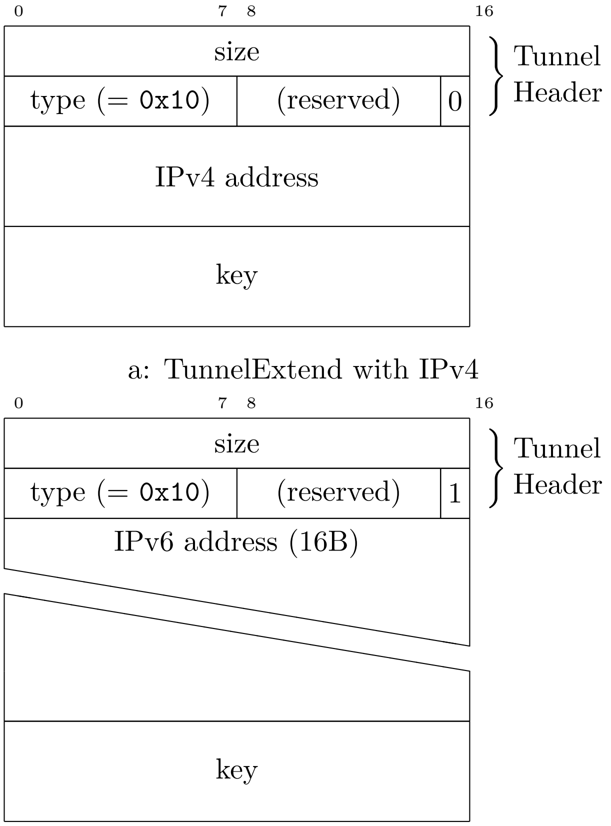
Tunnel-Level Packets

To transmit commands and messages via several hops, there are the tunnel packets Extend and Extended, Truncate and Truncated, Begin and End, Data, and Error. Tunnel packets either originate at or are routed to the tunnel controller. Before sending a tunnel packet, the sender adds sufficient randomly generated padding, so that packet and padding fit the fixed size requirement of the Circuit Opaque payload.

Therefore, to send a tunnel packet, the tunnel controller chooses a nonce and applies all the encryption layers using the negotiated ephemeral AES keys. Each tunnel packet features a digest field, that is set to zero before applying the encryption layers. When decrypting the Opaque payload, the hop can detect whether the packet is targeted to itself by testing the decrypted digest.

Tunnel Extension

In order to request an extension of the tunnel by one hop, the tunnel controller can send a Tunnel Extend (cf. figure 11) message to the final hop in the current tunnel. When receiving the Extend command, the final hop attempts to build a circuit to the peer whose address is part of the Extend message. The key for the Create handshake is also transmitted via the Extend message. The final hop is expected to parse the Created message and send the contained key and signature as the payload of a Tunnel Extended (cf. figure 12) packet to the tunnel controller. The tunnel controller can use these to verify the secret and derive a session key for communication with the newly appended hop.



a: TunnelExtend with IPv4

b: TunnelExtend with IPv6

Figure 11: TunnelExtend

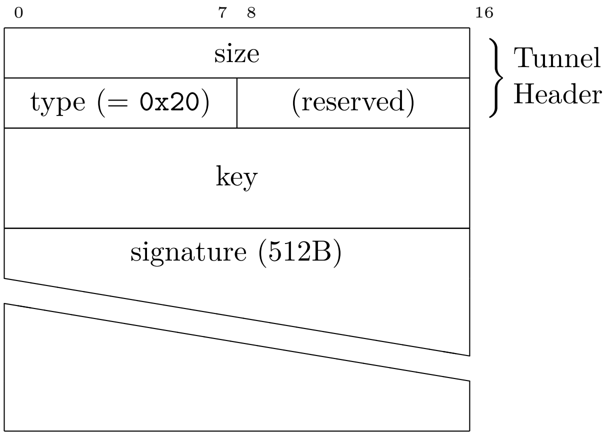


Figure 12: TunnelExtended

If an error occurs during the execution of the command, the final hop can send a Tunnel Error (cf. figure 13) reply back to the tunnel controller. This can be used to signal the tunnel controller that the newly requested circuit could not be constructed, but the tunnel itself is still working and does not need to be torn down. An error code is supplied to the tunnel controller to indicate the cause. The currently supported errors are:

Code	Name
`0x01`	`BRANCHING`
`0x02`	`PEER_UNREACHABLE`

The error `BRANCHING` signals, that the Extend command was send to an intermediate hop which already had an outgoing circuit as part of this tunnel. Branching tunnels are not allowed, therefore the Extend call is rejected. The error `PEER_UNREACHABLE` signals, that the Extend command could not be completed because the requested peer was unreachable. We might opt to adding additional error codes in order to improve the robustness of the implementation by allowing more fine-grained error-handling.

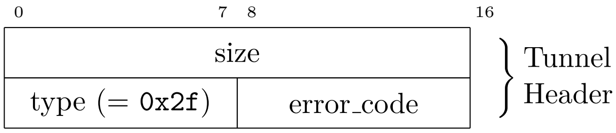


Figure 13: TunnelError

Tunnel Truncation

The Tunnel Truncate (cf. figure 14) message allows the tunnel controller to truncate parts of the tunnel. The receiving hop will tear down the outgoing circuit to the next hop using a Circuit Teardown message. On success, it will respond with a Tunnel Truncated (cf. figure 15) message. Similar to the tunnel extension, any errors will be communicated by sending a Tunnel Error (cf. figure 13) packet with one of the currently supported error codes:

Code	Name
`0x01`	`NO_NEXT_HOP`

The error `NO_NEXT_HOP` signals, that the Truncate command was send to the final hop of the tunnel and therefore no circuit could be truncated.

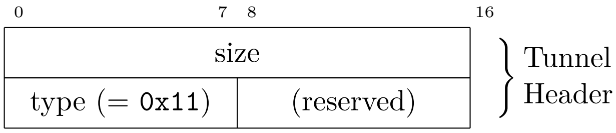


Figure 14: TunnelTruncate

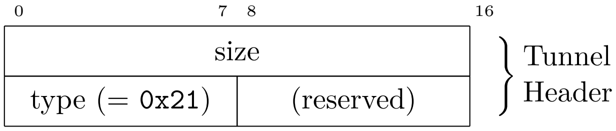


Figure 15: TunnelTruncated

Tunnel Data transmission

To use the tunnel for the transmission of data, the Tunnel Begin, Tunnel Data and Tunnel End packets are used. The tunnel controller may send a Tunnel Begin (cf. figure 16) message to the final hop in the tunnel.

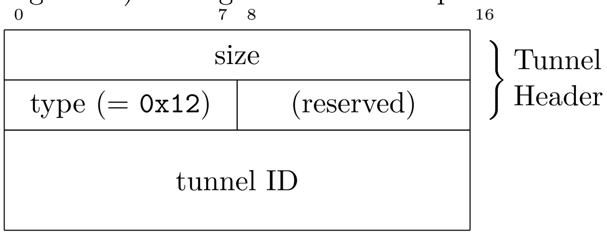


Figure 16: TunnelBegin

This signals to the peer, that it is supposed to listen for incoming data on the circuit, it received the Begin message on. It also signals that this peer is allowed to send Tunnel Data packets to the tunnel controller. The Begin message contains a field reserved for the tunnel ID, which is used to identify packets belonging to the same conversation.

After the Tunnel Begin message, both tunnel endpoints may send Tunnel Data (cf. figure 17) messages to each other.

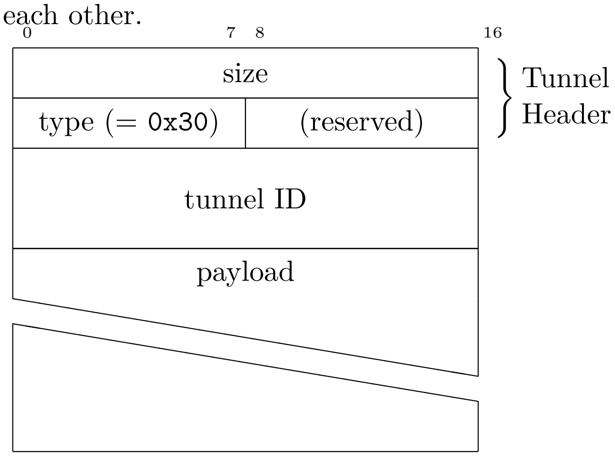


Figure 17: TunnelData

If any endpoint wants to explicitly end the conversation without tearing down the tunnel, the peer may send a Tunnel End (cf. figure 18) message and wait for the other peer to reply with a Tunnel End message. Any Data packets still buffered in the tunnel should be processed by the peer that initiated the Tunnel End until the Tunnel End reply is received.

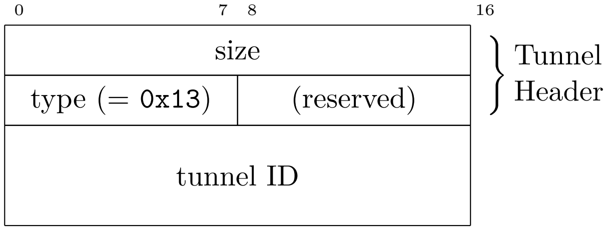


Figure 18: TunnelEnd

Conversations can theoretically be multiplexed via the same tunnel using different tunnel IDs, but we have opted to not utilize this possibility, since we only have one application (the VoidPhone) using the tunnels. During at least one active conversation, the tunnel controller is not allowed to send any other packets than the conversation related Begin, Data and End packets. The tunnel controller is however allowed to send Tunnel Truncate messages to any other hop in the tunnel, but in order to cause less disruption, it is encouraged to end all conversations with the final hop before truncating the tunnel. After both tunnel endpoints have negotiated the end of all conversations, the tunnel endpoint functions as a default final hop, so tunnel extension is possible, but not utilized by us.

Implicit Switch-overs

Since a conversation lifetime may surpass the anticipated tunnel lifetime, a conversation can be switched over to another tunnel. While peers are communicating over a tunnel with a certain ID, the tunnel controller may build a new tunnel between the peers and initiate a switch-over by sending a Tunnel Begin message with the same tunnel ID via the new tunnel. The receiving peer recognizes the tunnel ID coming from a different circuit, starts sending the Tunnel Data messages via the newly incoming tunnel and sends a Tunnel End packet via the old tunnel. The tunnel controller keeps monitoring the old tunnel for any remaining Data messages under the tunnel ID until the End message is received.

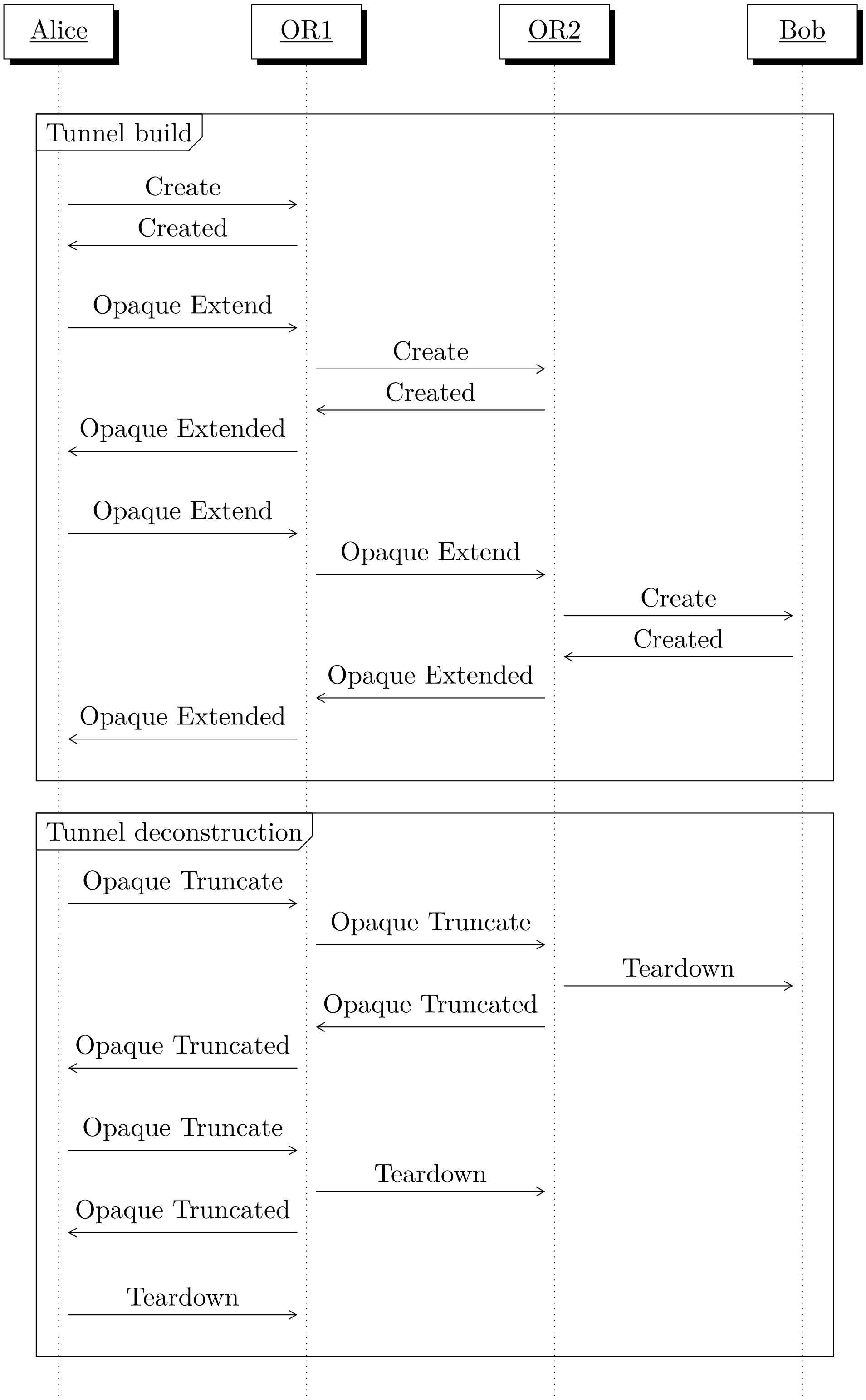


Figure 19: Tunnel construction and deconstruction

Security Measures

A core security requirement of onion tunnels is resistance against man-in-the-middle attacks. Randomly chosen peers which are used as hops may be malicious and try to gain sensitive information about the communicating parties. In particular an intermediate hop must not know about the origin or final destination of a tunnel, nor its position in the tunnel.

Connections from one hop to the next are secured with TLS encryption. To prevent leaking meta-data, all packets are padded to have a fixed size of 1024 bytes.

Building an onion tunnel requires iterative key exchanges while ensuring the correct identity of each peer. Otherwise a malicious peer may redirect an Extend request to an other attacker-controlled peer instead of the intended destination. This is achieved with signed handshake replies, which the TC can verify using an out-of-band acquired RSA public key.

To protect against spamming attacks, peers close circuits to other peers if there is no traffic on a circuit for one round time. Idle tunnels are not intended to exist since each hop has to keep track of the involved circuits and the circuit ID space is limited. This ensures that the circuit ID space is unlikely to be ever exhausted.

The connection from the TC to each hop is AES-encrypted with an unique session key, obtained from an ECDH key exchange. For all cryptographic operations the `ring` crate is used, which is based on BoringSSL – Google's fork of OpenSSL.

Peers deviating from the protocol are mandated to be disconnected. This means that peers sending invalid packet sizes or breach the protocol by sending invalid packet types, sizes or payloads are not allowed to be part of a tunnel. The tunnel controller or the hops immediately destruct the connection to the faulty peer in order to prevent any communication via broken peers that may potentially leak information or compromise anonymity. An unexpected teardown of a circuit which is part of a tunnel will be cascaded through the tunnel and the entire tunnel is destructed.

Future Work

Protocol performance under high load

In order to determine, whether we chose the right parameters for our peer-to-peer protocol, such as crypto algorithms, digest computation and packet lengths, we want to employ performance tests. The could be done in the form of benchmarks against Tor. We engineered our implementation in a way that it allows us to change these parameters easily, if desired.

Property based testing

In our original proposal, we included property based testing as a way to guarantee certain properties that our implementation has to fulfill. We may opt to implement the tunnel protocol implementation and error handling using property based testing. This will allow us to find loopholes and untreated corner cases in our peer-to-peer protocol implementation and weaknesses in our definition (like missing error cases).

Continuous Integration

It would be beneficial to make use of the CI/CD feature of GitLab, to automatically build and test the code on every commit to master. Additionally we would like have a high test coverage, which could also be reported as part of a CI pipeline.

Security vulnerability: Switch-over

Since the set of all tunnel IDs is used globally and not just between two peers, attackers may try to connect to a peer, that is the final hop in a tunnel and holding a conversation, to force a switch-over of the conversation to itself as destination. If the tunnel IDs are kept secret, this attack is very limited, since the attacker would have to guess 1 out of 2^{32} possible tunnel IDs.

Since the tunnel IDs in VoidPhone are directly taken from the API and the protocol can therefore not require them to be secret, there is a substantial risk that attackers may leak information or takeover conversations without the consent of the connected peers. We want to solve this by forcing the tunnel controller to sign the Tunnel Begin message using a non-brutable, randomly generated secret that needs to match each time the conversation is switched to a different tunnel.

Workload Distribution

While Oliver was especially concerned with error handling and the onion protocol, Dominik focused on implementing the API protocol and designing the concurrency model. However this is only a rough separation as we generally distributed the workload evenly among ourselves. Instead of designating exclusive areas of focus to each team member, smaller tasks were dynamically assigned on a by-need basis. This also ensured, that both of us maintained an overview of the project and were able to contribute to each part.

To put the effort spent in numbers, one can take the Git commit history into account, counting over 100 commits since May 8th. Some of larger refactorings were approached in pair-programming and thus were only committed from one account.