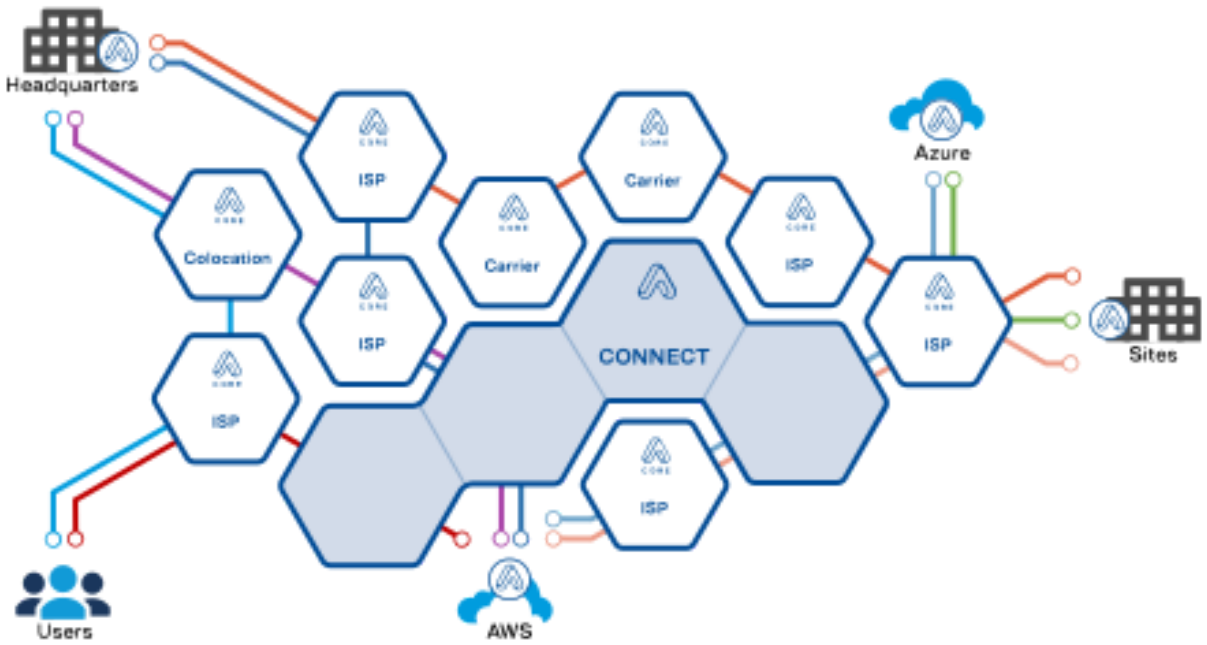
PATH LOOKUP PROTOCOL REPORT



Introduction

This report has been made for the **Security Verification and Testing** (**SVT**) Course at **Politecnico di Torino**. Its purpose is to provide all the necessary details to understand the modeling of the protocol analyzed, assumptions made, as well as a general understanding about SCION Network Architecture and its *Path Lookup Protocol*.

Background

In this section we are going to provide details about the network architecture and the description of the protocol analyzed. Understanding the architecture is necessary to deal with the protocol because it helps to clarify the purposes and the actors involved.

SCION Architecture

SCION stands for Scalability, Controllability and Isolation on Next generation Networks. It is a network architecture available from 2012 but still under development, designed and developed by ETH of Zurich.

The current Internet's core protocols, such as IP and BGP, have remained largely unchanged for decades, leading to various issues, including routing instability, lack of control over packet paths, and scalability challenges. Authentication is also lacking, relying on ad hoc solutions that are sensitive to compromise. Hence, SCION's goal is to design a highly available point-to-point communication infrastructure even in the presence of adversaries, addressing challenges such as on-path and off-path attacks. The architecture aims to provide transparency and control over forwarding paths, separating the control plane from the data plane, enabling multipath communication, and defending against network attacks. It also seeks transparency and control over trust roots, allowing users to select or exclude trust roots for authentication. Efficiency, scalability, and extensibility are vital, striving for high efficiency and improved scalability compared to the current Internet. The approach involves avoiding storing forwarding state on routers whenever possible, with end hosts assisting in network-layer functionality. Lastly, the architecture aims to support a global but heterogeneous trust environment, accommodating diverse legal jurisdictions and interests.

In this context we have to introduce the concept of **Isolation Domain** (**ISD**). Basically an ISD is a fundamental building block for achieving the properties of high availability, transparency, scalability, and support for heterogeneous trust. An ISD constitutes a logical grouping of **autonomous systems** (**ASes**). An ISD is administered by multiple ASes, which form the **ISD core**. We refer to these as **core ASes**. An ISD usually also contains multiple regular ASes. The ISD is governed by a policy, called the **trust root configuration (TRC)**, which is negotiated by the ISD core. The TRC defines the roots of trust that are used to validate bindings between names and public keys or addresses. Moreover, it can be considered as a **bootstrap to start all the other authentications**. The next picture helps to clarify these concepts just explained. Each of the colored bubbles identifies an ISD, while the gray areas are internal ISD cores.

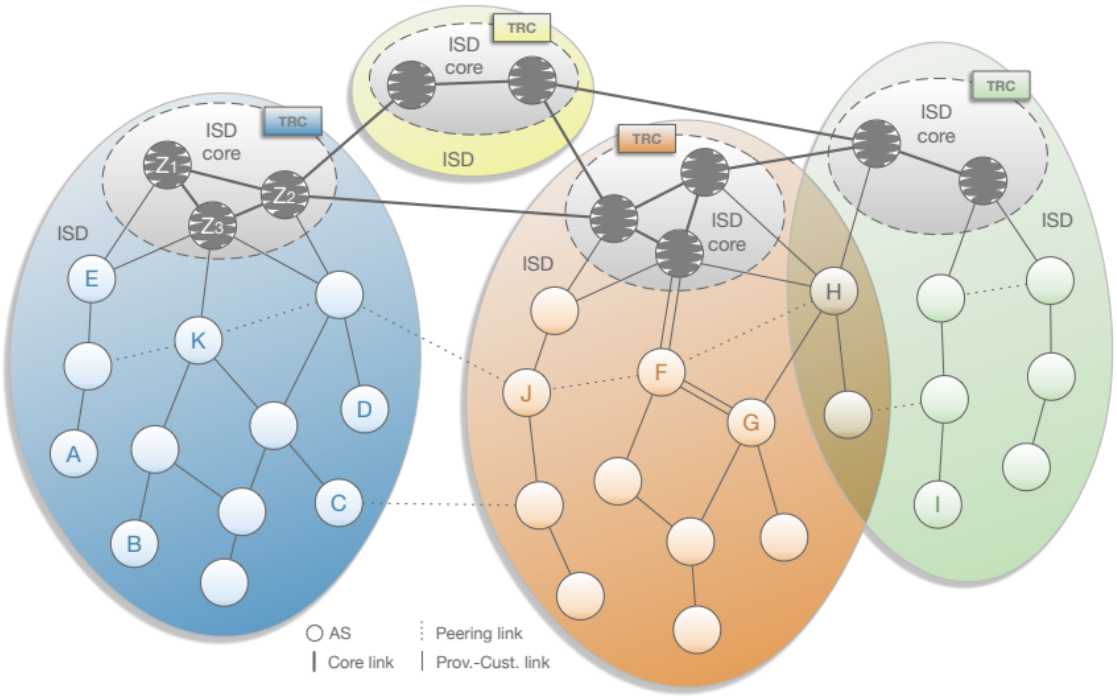


Figure 1: SCION Network example. ISDs are highlighted by different colors.

One of the most important features about SCION is its capacity to perform end-to-end path control. This means that a certain user, inside an AS belonging to an ISD, can send packets to another host in another domain choosing a specific path among those available. As we can see from the above picture, there are different link types in a SCION network:

- **Up-Segments**: these are provider-customer links, available from a customer source to a provider destination. An example from figure 1 is the link between the regular AS labeled as “K” and the core AS named “Z3”.
- **Core-Segments**: these are defined as core links. They identify a physical connection between a core AS in an ISD and another core AS, which can be both in the same ISD or in another one. An example from figure 1 is the link between “Z3” and “Z2”.
- **Down-Segments**: these are still provider-customer links but in the opposite direction, in which the source is the provider and the destination is the customer.

After this general understanding of the network, we can come up with the Path Lookup Protocol.

Path Lookup Protocol

The Path Lookup Protocol is one of the core protocols that implements path segments building and therefore, path selection. End-to-end communication is enabled by a combination of up to three path segments that form an end-to-end path. The goal of the path lookup process is to provide a source end host with diverse path segments and at least one set of connecting path segments, i.e., path segments that can be combined towards the destination by simply joining their corresponding endpoints. Depending on the location of source and destination end hosts, the path lookup process differs slightly.

To understand the protocol as a whole, we need to first look at the actors involved:

- **End Host (EH)**: it is the host starting the protocol and that requires the path segments to reach the destination. It starts the communication by first creating a TLS channel with its Local Path Server, and then issues a path request made up by the ISD number of the destination domain and the AS of the destination (basically, a request is like (ISD, AS)). This actor will also perform 2 other actions at the end of the protocol, when the request would have been processed. Indeed, it will verify the signatures of all the other actors and ask for the Trust Root Configuration (TRC).
- **Local Path Server (LPS)**: it is internal to the same AS of the EH and receives the request by the EH. Since path segments could be cached, it will check in its own cache to see if the segments are still available, otherwise it will forward the request to a Core Path Server. It will then wait for all the segments needed to reach the destination (up, core and down segments).
- **Core Path Server (CPS)**: this actor is placed in the same AS of the previous two but has many core links to other ISDs in the network. It makes the bridge that forwards the request to the Remote Core Path Server where the destination is placed. This actor will then wait for core and down segments to forward to the local path server, adding also the up segments to reach itself.
- **Remote Core Path Server (RCPS)**: this has essentially the same role as the previous one but in another domain. It is in charge to forward the request to the destination and then waits to receive the down segments to reach it.

Following is provided an example describing a protocol flow in which the destination is outside the source ISD.

Example:

In this example, we assume that the desired paths are not yet cached in the path servers. First, an **end host** from **AS (1,10)** *(which means ISD 1, AS number 10)* that wishes to contact a host from **AS (2,23)** contacts its own **local path server**, requesting path segments connecting source (1,10) with the destination (2,23). The local path server, using an up-segment, contacts a **core path server inside the local ISD** *(i.e., with AS (1,1))*, requesting path segments from the core path server's AS (1,1) to the destination AS (2,23). *(Note that the local path server postpones this request if it has no up-segment. In this case, another protocol will perform this action.)*

The core path server of (1,1) takes any core-segment to **another core AS from the destination ISD 2** *(basically, the actor on the right side is the RCPS with AS (2,2))*, and queries this AS's path server. *(As before, this is postponed until the core-segment is available.)*

In our example, the path server in AS (2,2) is asked about down-segments of the destination AS (2,23). We emphasize that the down-segments of AS (2,23) do not have to originate necessarily from AS (2,2), they can originate from any other core AS from ISD 2. As soon as the core path server from ISD 2 has appropriate down-segments, it will return up to **K** of them to the core path server in AS (1,1), which verifies the path segments. Moreover, this entity can query again the remote core path server to get all the certificates, owned by ASes signing the segments, or the TRC if locally cached information is missing or outdated. This process can be done with the exchange of TRC request (or cert. request) and then with the response binded to the request.

Next, this core path server *(the local one to the source)* has to find up to **K** core-segments between its AS (1,1) and ASes that originated the received down-segments. At least one such core-segment has to be found, otherwise the path server waits for it. Then, down-segments with the corresponding core-segments are returned to the local path server of AS (1,10) *(which verifies the path segments as well and can query the local core path server for certificates or TRC)*. The local path server adds up-segments to the set of obtained paths, adds additional core-segments *(if they are cached)* connecting up- and down-segments, and sends the entire response to the end host. The up-segment to the

core AS (1,10) has to be within this response. Finally, the end host verifies the received path segments.

Figure 2 depicts all the steps described above.

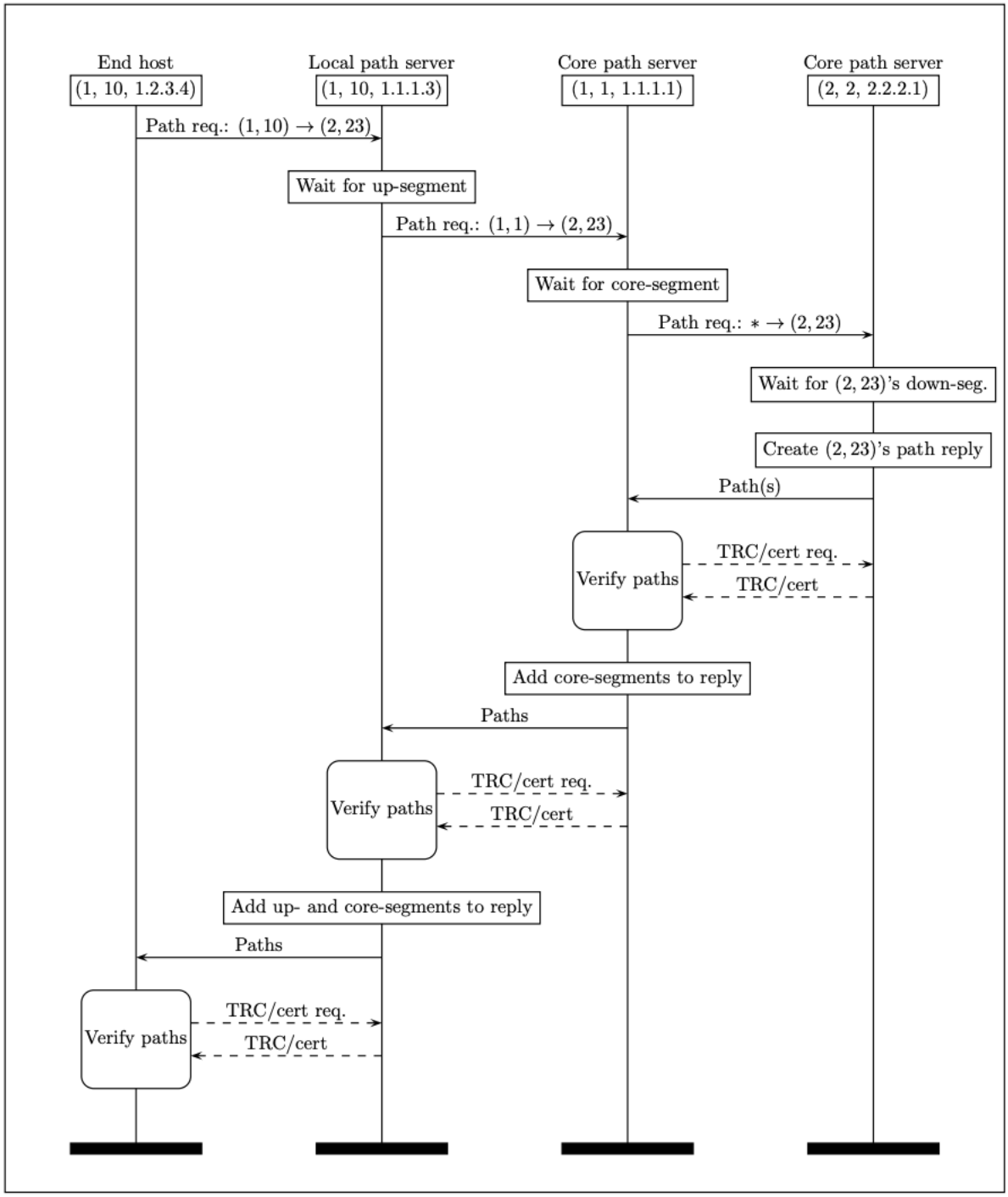


Figure 2: Path Lookup Protocol Description

Path Lookup Protocol - Modeling

We are now ready to make a model of the protocol, hence we will discuss the goals of this analysis, the assumptions and simplifications that were made, as well as their impact on the reliability of the analysis. We will then move to the protocol specification in terms of Proverif script.

Goal of this analysis is to formally verify a protocol which is already in production, in order to verify that message forging and signature compromission cannot be performed, unless under some specific conditions.

Assumptions and Simplifications

To model the protocol we have made some assumptions necessary to lighten the analysis and to fully describe it regardless of Proverif limits. We will now explain these assumptions:

- The original protocol gives the possibility to have many other actors on the way, basically they are other ASes on the way to reach the destination. When an AS is traversed by a packet, it adds its entry (by signing it) to the response in order to provide the whole path segment. In our model there are no other intermediaries on the path, the source knows that the only other actors are LPS, CPS, RCPS and the destination.
    - **Why:** because otherwise I was not able to model a while loop to verify each signature and the protocol would have been heavier to analyze.
- A simplification in the Segments structure has been adopted, now segments are defined as a triple (AS identifier, signature, public key) - This impacts the analysis in case there is a revocation, because we are not providing the certificate itself.
    - **Why:** since we are already wrapping objects one into another and creating lists, it would have been really hard to unwrap all of them, including the certificate itself. Moreover, the certificate revocation is performed by another protocol and it would have added another complexity level.
- After the signature verification, we should verify that not only the signatures are valid but also that the authority who issued them is a trusted one. In our case, TRC is

only requested and “externally” verified. This step is seen as a black box that cannot fail.
    - **Why:** in the protocol there is no description about how the root certificate is verified or how the trust root configuration is designed, hence I modeled it as if it is just an event, always successful.
- In the original protocol everyone should be able to verify the signature but in our case we made this possible only for the EH.
    - **Why:** it would have been heavy to always perform the signature verification in each stage of the protocol. This assumption only makes the protocol slower in case the verification fails, because it is not stopped before but only at the end. In any case this verification by the other actors is not mandatory in the protocol.
- The AS destination is “assigned” the AS identifier by the caller. We then will assume that the caller knows which AS corresponds to that identifier and directly forwards the request to it.
- Channels have been implemented in half duplex to avoid reading or writing problems.
    - **Why:** There is misbehavior leading to a property that cannot be proved. The result would have been a non termination of the formal verification. Certainly, this assumption has no limitations over the assessment process since 2 half-duplex channels are equivalent to a full-duplex one.
- Since all the messages are exchanged on a secure channel that provides an authentication mechanism, indeed SCION uses a combination of ARPKI and PoliCert to manage certificates and TLS for the communication (as stated in the SCION book at pages 86-89), we modeled the channels as private channels.

Protocol and Properties Specification

The protocol has been then modeled using 5 sub processes identifying the 4 entities described and the destination to be reached. In addition, a further sub process has been defined in order to perform signature verification from the side of an End Host. The interactions have been modeled by the usage of concurrency (e.g. pA | pB) and replication (e.g. !pA | !pB). Keys generation was modeled using a key material that binds the public and

private keys. There are 12 monodirectional private channels on which the requests and responses are sent. Many queries have been defined to verify the reachability of every event defined within the protocol and there are also some correspondence queries to verify that some events occur only if another event has occurred. In the end, some other checks are performed to verify that the entries of a path segment have not been modified or undesirably created unless a private key has been discovered by an attacker. This is performed by checking that a specific signature has been verified only if the creation event occurred with a private key corresponding to the public key used for the verification.

Adding some details, the following properties have been queried:

```
(*Query definition*)
(*Simply sanity checks for basic events - Reachability*)
query isd_as: ISD_AS; event(send_request(ch_source_local_server, isd_as)). (*Expected to be true*)
query isd_as: ISD_AS; event(send_request(ch_local_server_core_server, isd_as)). (*Expected to be true*)
query isd_as: ISD_AS; event(send_request(ch_core_server_core_remote_server, isd_as)). (*Expected to be tr
query isd_as: ISD_AS; event(send_request(ch_core_remote_AS, isd_as)). (*Expected to be true*)
query event(destination_received_request()). (*Expected to be true*)
query pk: public_key; event(signature_done_by_AS(pk)). (*Expected to be true*)
query updated_list: as_entry_list; event(send_segments(ch_core_remote_server_core_server, updated_list)).
query as_list: as_entry_list; event(core_server_receive_path(as_list)). (*Expected to be true*)
query trc: trust_root_configuration; event(trc_send_by_remote(trc)). (*Expected to be true*)
query trc: trust_root_configuration; event(trc_verified_by_core_server(trc)).
query trc: trust_root_configuration; event(trc_send_by_core_server(trc)).
query trc: trust_root_configuration; event(trc_verified_by_local(trc)).
query trc: trust_root_configuration; event(trc_send_by_local(trc)).
query trc: trust_root_configuration; event(trc_verified_by_eh(trc)).
query as_list: as_entry_list; event(end_host_receive_paths(as_list)). (*Expected to be true*)
query as_list: as_entry_list; event(local_server_receive_path(as_list)). (*Expected to be true*)
query entry: as_entry; event(remote_core_server_receive_path(entry)). (*Expected to be true*)
query pk: public_key; event(signature_done_by_AS(pk)). (*Expected to be true*)
query pk: public_key; event(signature_perfomed_by_core_remote_server(pk)). (*Expected to be true*)
query pk: public_key; event(signature_perfomed_by_core_server(pk)). (*Expected to be true*)
query pk: public_key; event(signature_perfomed_by_local(pk)). (*Expected to be true*)
query pk: public_key; event(dest_signature_verified(pk)). (*Expected to be true*)
query pk: public_key; event(remote_signature_verified(pk)). (*Expected to be true*)
query pk: public_key; event(core_signature_verified(pk)). (*Expected to be true*)
query pk: public_key; event(local_signature_verified(pk)). (*Expected to be true*)
```

All the queries above are used for reachability of the single events involved in the protocol. The **first 5 queries** can be seen as belonging to the same logical group, it is the verification of the **request forwarding mechanism** and its verification from each process involved. The **second group** is made of queries **checking that the TRC** is correctly sent and verified

by each actor. Then, there is the **group** composed of all the queries that **check the signatures execution** and their verification.

Other than these sanity checks, there are some security properties used to verify the behavior of the protocol and that the occurrence of a specific event is related to the occurrence of a previous event. The aim here is to check that an attacker is not able to manipulate the execution of events and, hence, the order of messages exchanged.

```
(*Event Correspondence, sanity checks for the workflow of the protocol*)
query isd_as: ISD_AS; event(send_request(ch_local_server_core_server, isd_as)) ==> event(send_request
(ch_source_local_server, isd_as)). (*Expected to be true*)
query isd_as: ISD_AS; event(send_request(ch_core_server_core_remote_server, isd_as)) ==> event(send_request
(ch_local_server_core_server, isd_as)). (*Expected to be true*)
query isd_as: ISD_AS; event(send_request(ch_core_remote_AS, isd_as)) ==> event(send_request
(ch_core_server_core_remote_server, isd_as)). (*Expected to be true*)
query isd_as: ISD_AS; event(destination_received_request()) ==> event(send_request(ch_core_remote_AS,
isd_as)). (*Expected to be true*)
query paths: as_entry_list; event(send_segments(ch_core_remote_server_core_server, paths)) ==> event
(destination_received_request()). (*Expected to be true*)
query paths: as_entry_list; event(core_server_receive_path(paths)) ==> event(send_segments
(ch_core_remote_server_core_server, paths)). (*Expected to be true*)
query tr: trust_root_configuration; event(trc_verified_by_core_server(tr)) ==> event(trc_send_by_remote(tr)).
(*Expected to be true*)
query tr: trust_root_configuration; event(trc_verified_by_local(tr)) ==> event(trc_send_by_core_server(tr)) .
(*Expected to be true*)
query tr: trust_root_configuration; event(trc_verified_by_eh(tr)) ==> event(trc_send_by_local(tr)) .
(*Expected to be true*)
```

The **first 6 queries** verify that each **sending event of the request** is due to a previous sending from the “previous actor” in the protocol. Basically, by specifying them we are querying if the sending events are binded to the original request made by the **End Host**. For instance, the send request from a local server determines a previous occurrence of the send request from the end host, as well as for the other sendings *(the local server for the core server and the core server for the remote core server)*. Clearly, the **remaining 3 queries** (belonging to this group) checks that all the **TRCs verification** can occur only after the receipt of the TRC itself.

Last queries performed deal with the signatures verification, their success determines the authentication of the received segments. Hence, the verification of each of the signatures using the provided public key *(for: destination, remote core server, local core server and local path server)* imply that previously there was a signature performed by each of these entities

using the associated private key to the public key provided. The only way this should not succeed is if the attacker was able to discover the corresponding private key.

```
(*Signature Verification*)
query k: keymat; event(dest_signature_verified(gen_public_key(k))) ==> event(signature_done_by_AS
(gen_public_key(k))) || attacker(gen_private_key(k)). (*Expected to be true*)
query k: keymat; event(remote_signature_verified(gen_public_key(k))) ==> event
(signature_perfomed_by_core_remote_server(gen_public_key(k))) || attacker(gen_private_key(k)). (*Expected to
be true*)
query k: keymat; event(core_signature_verified(gen_public_key(k))) ==> event(signature_perfomed_by_core_server
(gen_public_key(k))) || attacker(gen_private_key(k)). (*Expected to be true*)
query k: keymat; event(local_signature_verified(gen_public_key(k))) ==> event(signature_perfomed_by_local
(gen_public_key(k))) || attacker(gen_private_key(k)). (*Expected to be true*)
```

Indeed, the queries below are checking, respectively:

- **The verification of the first signature** using the public key provided by the destination, is successful only if the destination itself previously performed the signature using the corresponding private key.
- **The verification of the second signature** using the public key provided by the remote core path server, is successful only if the remote core path server previously performed the signature using its own private key (corresponding to the public one provided).
- **The verification of the subsequent signature** using the public key provided by the core path server (the local one to the source), is successful only if the core path server previously performed the signature using its own private key (corresponding to the public one provided).
- Finally, **the verification of the last signature** using the public key provided by the local path server, is successful only if the local path server previously performed the signature using its own private key (corresponding to the public provided).

There is only one case in which these verification are successful even if there was no signature performed by one of the processes and it is in case the private key is disclosed,

because in that case the attacker would be able to perform the signature in place of the process.

Results

Even with some limitations, overall the formal verification is successful. All the events are successfully reached and the security properties verifying the protocol behavior determine that all the messages are exchanged in the right order. Moreover, the signature verification is the key point of the formal verification, since the authentication of all segments received by the EH relies on that. From our verification, the protocol results to be correctly designed and the signatures are not attackable (unless the disclosure of the private keys from the attacker). Anyway, due to the many simplifications, further analysis is needed to fully verify the protocol. Indeed, components like: TRC Verification, the Beacon Service that provides the segments and the protocol for certificate revocation, must be modeled and verified.

Sources

These are all clickable links with resources used to develop this document and the analysis:

- Scion Book - Scion Architecture Book
- Proverif - Manual
- SVTProject - Github Repository