SCION's Hidden Paths Design Formal Security Analysis

Author(s):
Zenoni, Mauro

Publication Date:
2020

Permanent Link:
https://doi.org/10.3929/ethz-b-000411121 →

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SCION's Hidden Paths Design Formal Security Analysis

Master Thesis

Mauro Zenoni

April 7, 2020

Professor: Prof. Dr. David Basin
Supervisors: Dr. Ralf Sasse, Dr. Jonghoon Kwon

Department of Computer Science, ETH Zürich

Abstract

When designing complex software architectures, shortcomings concerning its intended security properties are simply to be expected. Fortunately, nowadays we can rely on sophisticated tools to support the formal modeling and analysis of security protocols, which can help to drastically reduce the possibility of real-world attacks on the resulting infrastructure. More precisely, with the use of automated provers, we can obtain formal proofs for all the specific security properties that we are interested in verifying for the message exchanges under analysis. In this thesis, we focus on SCION, a security-first Future Internet architecture developed at ETH Zurich. In particular, we start from an intuition on how to address a lack of server authentication in SCION's Hidden Paths Design and, from there, we propose a sensible solution to this problem that maximizes the reuse of well established protocols and minimizes the deployment efforts, all while making sure that we are not violating SCION's architectural principles. Finally, we perform a formal security analysis of our solution by leveraging Tamarin, a state-of-the-art security protocol verification tool that comes with a sound and complete proof search algorithm. In this context, we also explain how our analysis successfully yielded all the results we were hoping for, thus confirming the correctness of our proposed solution and justifying its suitability to be adopted as part of SCION's specification.

Contents

Contents **iii**

1 Introduction **1**

2 SCION Internet Architecture **5**
2.1 Architecture Overview 5
2.2 Control Plane 7
2.3 Data Plane 8
2.4 Control Plane Authentication 9
2.5 Hidden Paths Design 12
2.5.1 Hidden Paths Infrastructure 12
2.5.2 Hidden Paths Server Authentication 13

3 Tamarin Prover **15**
3.1 Security Protocol Model 16
3.1.1 Terms 17
3.1.2 Rules 17
3.1.3 Facts 18
3.1.4 Security Protocol Theory 19
3.2 Adversary Model 20
3.3 Security Properties 21
3.4 Restrictions 21
3.5 Proof Search 21

4 Problem Statement and Proposed Solution **23**
4.1 Intuition 25
4.2 Assumptions 26
4.3 Realization 27
4.4 Rationale for the Proposed Solution 28

5 Reissuance Protocol **31**

5.1 Reissuance Protocol Specification 31
5.1.1 Initial Knowledge 33
5.1.2 Message Exchange 33
5.2 Reissuance Protocol Extension 37
5.3 Modeling Choices and Abstractions 38
5.3.1 Signatures 38
5.3.2 Digital Certificates 38
5.3.3 Abstracted SCION Topology 39
5.3.4 Abstracted TRC 39
5.3.5 SCION Certificates Dissemination 40
5.3.6 Timestamps 42
5.3.7 Abstracted SCION Certificates Structure 43
5.4 Modeled Protocol Specification 44
5.4.1 Infinite Roles 45
5.4.2 Model Splitting 46

6 Proof Summary **47**
6.1 Adversary Model 47
6.2 Security Properties 48
6.2.1 Agreement Properties 49
6.3 Proofs and Results 54

7 Conclusion **59**

Bibliography **63**

Abbreviations **67**

Glossary **69**

A TRC Structure **71**
A.1 The `validity` object 72
A.2 The `attributes` object 72
A.3 The `keys` object 72
A.4 The `signature` object 72

B SCION Certificates Structure **73**
B.1 The `validity` object 74
B.2 The `keys` object 74
B.3 The `issuer` object 74

C AS Certificate Reissuance Protocol **75**
C.1 Initial Knowledge 75
C.2 Full Formal Specification 76

D Abstracted Reissuance Protocol **81**
D.1 Initial Knowledge 81
D.2 Full Formal Specification 82

Chapter 1

Introduction

Traditionally, security analyses of cryptographic protocols have been conducted by providing informal arguments for their presumed correctness. Such arguments have oftentimes been proven to be too weak or even flawed, thus failing to foresee real-world attacks on protocols otherwise deemed trustworthy. Nowadays, we can instead rely on powerful tools to support the formal modeling and analysis of security protocols. Especially, with the use of automated provers, we can obtain formal proofs for all the specific security properties that we are interested in verifying for the message exchanges under analysis. For these purposes, in the course of this thesis we will leverage *Tamarin* [3], a state-of-the-art security protocol verification tool which comes with a sound and complete proof search algorithm.

More precisely, the security analysis that we are going to perform concerns *SCION* [2], a security-first Future Internet architecture developed at ETH Zurich and designed for high-availability and efficient packet delivery, even in the presence of malicious operators and devices actively corrupting the network. This network architecture leverages the *Isolation Principle*, which aggregates different *Autonomous Systems (ASes)* deploying SCION into interconnected *Isolation Domains (ISDs)*, each one protected from external influences and maintaining independent roots of trust.

Now, let us imagine coming up with an idea on how to solve the latest issue with the design of a complex software architecture. What is more, let us suppose that the problem to be addressed concerns a security critical aspect of the system under development. How to be sure, then, that a solution based on mere intuitions can effectively be implemented without compromising the overall security of the resulting system?

This is exactly the question that we had to ask ourselves after the engineers at *Anapaya Systems* [1], a spin-off of ETH Zurich leading the implementation of the SCION architecture for the consumer market, brought to our attention the existence of a critical authentication problem in SCION's specification. In particular, we came upon a lack of server authentication in the *Hidden Paths Design* [17], whereas, in that context, a *Beacon Server (BS)* and a *Hidden Path Server (HPS)* located in different ASes are supposed to only communicate over a secure channel, i.e., authenticated at both ends and confidential. Hence, the need to provide some assurance over the idea of bootstrapping TLS trust directly from SCION's *AS Certificates*, so as to solve this security shortcoming by establishing an encrypted QUIC [9] channel between the two servers. That entails both finding a concrete solution which realizes this intuition and carrying out a formal security analysis of the resulting protocol to verify its legitimacy.

Contributions

We expect our work to be of particular interest to the people at Anapaya Systems, to the scholars of the Network Security group, and to future users of Tamarin. In particular, throughout this thesis we will present a variety of achievements encompassing different topics concerning the field of information security. Our work spans discussions about the security of SCION's architecture, considerations on design choices in the context of Hidden Path communication, symbolic protocol modeling and formal security analysis in Tamarin.

Below, we are going to provide a summary of our main contributions.

- We have investigated the security problem afflicting the Hidden Paths Design and expanded our inquiry to the broader lack of server authentication mechanisms in SCION.
- We have found a solution implementing the idea of bootstrapping TLS trust from AS Certificates. When coming up with a concrete protocol realizing this intuition, we focused on ease of deployability. Hence, we made the effort to reuse well established mechanisms already implemented in SCION and to only require minimal changes to the existing SCION infrastructure.
- We have modeled in Tamarin both the *Reissuance Protocol*, i.e., the protocol in SCION that we have based our solution on, and our proposed extension for it. In this phase, we had to make some interesting modeling choices to effectively reproduce the Reissuance Protocol in Tamarin.
- We have performed a formal security analysis of the Reissuance Protocol in Tamarin, thus providing strong assurance of its security properties. Given the successful results from the Tamarin prover, we were able to conclude that our proposed extension for the Reissuance Pro-

tocol constitutes a suitable solution to the authentication problem in the Hidden Paths Design.

Related Work

For our security analysis, we decided to rely on the Tamarin prover. Other protocol verification tools, such as *ProVerif* [6] and *Scyther* [7], already enjoyed great popularity when Tamarin was first introduced in a 2012 paper [15] from Schmidt et al. However, Tamarin presents itself as a more modern alternative, capable, among other things, of providing fully automated support for handling Diffie-Hellman exponentiation [15] and for verifying observational equivalence properties [5]. The reason we lean on this particular tool for the purposes of our thesis, though, is that it allows us to make use of AC-operators [14], including multisets and natural numbers, to specify temporal properties in an intuitive way, and to simultaneously deal with an unbounded number of sessions and a mutable global state [4]. This combination of features sets Tamarin apart from its predecessors and, as we will see in later chapters, is of primary importance for modeling the protocols that will be considered in our security analysis.

The solution we have designed to bootstrap a QUIC channel between a BS and an HPS, while relevant today and much requested by the engineers at Anapaya Systems to deploy a first fully working implementation of the Hidden Paths Design, could be replaced in the future by *PISKES* [13]. This is a proposed extension to the SCION architecture that refines and expands the *Dynamically Recreatable Keys (DRKey)* system. DRKey was originally part of SCION's specification [12], but it was later set aside because of some vulnerabilities caused by an improper application of signature schemes that were recently discovered by leveraging the Tamarin prover [10] .

PISKES is a symmetric-key derivation system which assumes the prior existence of an asymmetric key pair for each of the participating ASes in SCION. The corresponding public key has to be authenticated directly by SCION's *Control Plane PKI (CP-PKI)*. We have listed this key as the *Encryption Key* in the AS Certificate format specification in Section B.2 of the Appendix. All ASes that want to deploy PISKES also need to set up an infrastructure of dedicated PISKES key servers. These key servers, among other things, are responsible for exchanging with other ASes the symmetric root keys which will later be used to derive a hierarchy of symmetric keys for all the entities located within these ASes. This initial key establishment is carried out between pairs of ASes and is secured via their Encryption Keys.

This symmetric-key derivation system implements full end-to-end authenticity and secrecy of communication between two entities located in different ASes in SCION, provided that these source and destination ASes are honest. In particular, while for the solution proposed in this thesis we assume the existence of a trusted local *Certificate Authority (CA)* service in the source and destination ASes, for PISKES we need to assume that their respective key servers are not compromised, or otherwise an attacker could impersonate any entity located within these ASes. Under this assumption, PISKES enables a BS and an HPS located in different ASes to obtain a *Pre-Shared Key (PSK)* [19] which can be directly used to bootstrap an encrypted QUIC channel connecting the two.

Outline

Preliminary to the understanding of this thesis are some fundamental notions on Tamarin and on SCION. These topics will be fully covered in the next two introductory chapters. In the fourth chapter, we will illustrate step by step how our attempt to answer a request for help from the engineers at Anapaya Systems evolved into the solution that we ended up modeling in Tamarin. In chapter five, we will presents all the relevant details of the Reissuance Protocol and the main modeling choices needed to reproduce it effectively in Tamarin. In the sixth chapter, we will introduce the adversary model we assumed in our security analysis, the security properties we chose to verify, and the formal results we were able to obtain from the Tamarin prover. Finally, in the very last chapter, we will draw the final conclusions on our work.

All the Tamarin models that we have produced in course of this thesis and all the formal proofs of the security properties that we have verified for them have been made available for consultation[1].

[1]Tamarin Models and Proofs:
https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/scion-hps-zenoni.zip.

Chapter 2

SCION Internet Architecture

SCION is a Future Internet architecture designed for high-availability and efficient packet delivery, even in the presence of malicious operators and devices actively corrupting the network. As of 2020, SCION is an on ongoing project at ETH Zurich. This architecture was extensively documented in a 2017 book [12] from the Network Security group. More up to date documentation over specific processes in SCION can be found in the GitHub repository[1] where the project is maintained.

2.1 Architecture Overview

The fundamental building block of the entire SCION architecture is what we call *Isolation Domain (ISD)*, i.e., a logical grouping for *Autonomous Systems (ASes)*. A privileged set of these ASes that we call *Primary ASes* constitutes the *ISD Core* and carries out all administrative tasks within the ISD.

Each ISD is regulated by its own *Trust Root Configuration (TRC)*, which is defined and maintained by the ISD Core and contains a list of all Primary ASes in the ISD, as well as some public keys which serve as the *roots of trust* for all *SCION Certificates* from the same ISD. Policies governing the use and update of these keys are also encoded within the TRC itself. Since authentication in SCION strictly relies on SCION Certificates and these certificates are authenticated by the ISD's roots of trust, the TRC can be seen as the sole way for bootstrapping all authentications for ASes within the same ISD.

In a process called *beaconing*, Primary ASes make use of *Path-segment Construction Beacons (PCBs)* to explore possible inter- and intra-ISD routing paths. As they traverse the network, these *beacons* collect path information that they accumulate inside cryptographically protected fields called *hop fields* and progressively build up *path segments*. A few PCBs are later selected and

[1]`SCION` codebase: `https://github.com/scionproto/scion`. Accessed: 2020-02-25

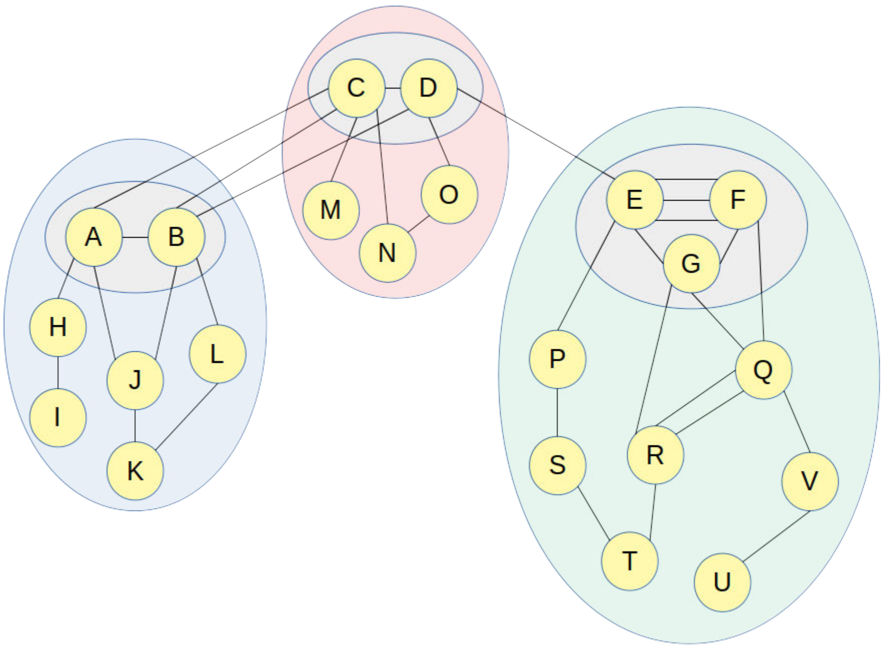

Figure 2.1: Simple SCION Architecture Example. We can distinguish three ISDs, each one containing several interconnected ASes. ISD Cores are represented at the top of the ISD. The Primary ASes are respectively A and B for the first ISD, C and D for the second ISD, and E, F and G for the third one.

their path segments registered at dedicated servers, thus making the discovered routing paths available to end-hosts. These phases of *path exploration* and *path registration* constitute SCION's Control Plane architecture and will be further discussed in the next section.

In SCION, we distinguish between *border routers* and *internal routers*. While border routers are meant for connecting ASes together, internal routers only forward packets within a single AS. A fundamental paradigm of the SCION Architecture is that packets sent through the network already carry with them all AS-level routing information that border routers use to direct them towards their destination via the desired sequence of ASes. This paradigm, usually referred to as *packet-carried forwarding state*, completely removes the need for stateful inter-AS forwarding tables at the border routers, thus establishing a more efficient stateless forwarding plane.

In order to obtain the end-to-end path required for forwarding a packet to its destination, an end-host must first retrieve the necessary path segments via *path lookup* and combine them appropriately via *path combination*. These two processes, which together are referred to as *path resolution*, constitute SCION's Data Plane architecture and will be explained in further detail in Section 2.3.

The main components of SCION's AS infrastructure are:

- *Beacon Servers*: they are responsible for the beaconing process and hence for the periodic creation and propagation of PCBs.
- *Path Servers*: they are organized in a hierarchy of caches, and are responsible for registering path segments provided by beacon servers and for making them available to the end-hosts.
- *Certificate Servers*: they are responsible for managing the cryptographic keys of an AS and for caching TRCs and SCION Certificates.

2.2 Control Plane

While inter-ISD beaconing allows Primary ASes to discover paths to other Primary ASes from different ISDs, unprivileged ASes learn paths to their ISD Core via intra-ISD beaconing.

In particular, in SCION the following path segments can be assembled:

- *Up-Segment*: a path segment from an unprivileged AS to its ISD Core.
- *Down-Segment*: a path segment from the ISD Core to an unprivileged AS in the same ISD.
- *Core-Segment*: a path segment between Primary ASes.

Typically, path segments are bidirectional, meaning that mirroring up- and down-segments can be converted and used interchangeably.

Beaconing is a periodic and asynchronous process which originates in the ISD Core, where *Core Beacon Servers* generate PCBs. For discovering up- and down-segments, these beacons are then propagated in a multipath flood to downstream ASes, while for learning core-segments, they are spread throughout the entire network traversing only Primary ASes.

At every AS, a beacon server adds to the incoming PCB a new *hop field* before forwarding it to the next AS. This hop field contains the identifiers for the *ingress* and *egress* links of the AS, thus allowing for a fine-grained representation of paths which distinguishes between disjoint paths even for the same AS sequence. For example, note in Figure 2.1 how, in the rightmost ISD, ASes R and Q, or even Primary ASes E and F, are connected via more than one direct link.

Each hop field is cryptographically protected in order to ensure path correctness. In particular, an AS adding this forwarding information to a PCB, makes sure to also compute a *message authentication code (MAC)* over it, using a secret symmetric key only known to its own beacon servers and border routers. This MAC can be efficiently verified by the border routers of the same AS during package forwarding. Moreover, each AS uses its private *Signing Key* to sign all PCBs that it forwards. In Section 2.4 we will see how these signatures can be used by all entities to verify the authenticity of a PCB or of the corresponding path segment.

Every time a beacon server receives a PCB, it can decide to extract its path segment and register it at an appropriate path server. SCION distinguishes between *core path servers*, which are located inside the ISD Core, and *local path servers*, which reside in unprivileged ASes. More precisely, down-segments and core-segments are registered at core path servers, while up-segments are only registered at local servers.

2.3 Data Plane

We have seen in the previous section how up-, down-, and core-segments are discovered and registered at appropriate path servers across the SCION network.

Now, by referring to Figure 2.1, let us consider a source end-host located in AS K, inside the leftmost ISD, wanting to deliver a packet to a destination end-host located in AS R, inside the rightmost ISD. In order to do so, K needs a valid path from source to destination, that he is going to obtain by combining an up-segment to the source ISD Core, a core-segment between source and destination ISD Cores, and finally a down-segment from the destination ISD Core to AS R.

The source end-host can fetch all required path segments by querying a local path server. First, this local path server responds with an up-path to one of the Primary ASes in the same ISD, e.g., a path from K to B. Then, if not already present inside its cache, the local path server will contact a Primary AS in the same ISD for the remaining core- and down-segments. In turn, this core path server will query a core path server in the destination ISD Core if the requested path segments were not already cached locally. Eventually, the local path server will update its cache with the newly retrieved core- and down-segments and serve them to the source end-host. In our running example, a valid core-segment could describe a path from B to E, and a down-segment a path for reaching R from E.

Once the end-to-end forwarding path is assembled, this is added to the header of the SCION packet, which is now ready to leave AS K. SCION border routers will take care of forwarding the packet to the next AS following the links specified in its header. Each AS involved in the packet delivery can verify the authenticity of this forwarding information by checking the MAC that was previously computed in the beaconing phase.

All along the network, the packet is efficiently forwarded without the need for intermediate border routers to inspect the address of the destination end-host in R nor to consult any inter-AS routing table. Only border routers in AS R will read the destination address to deliver the packet to the correct local end-host. It is worth making clear that SCION does not define the protocols used for intra-AS communication. Hence, an AS is free to choose independently the technology used for the delivery of packets between the end-hosts and their respective border routers.

2.4 Control Plane Authentication

We have previously mentioned how all authentications in SCION are based on SCION Certificates. In particular, SCION's *Control Plane PKI (CP-PKI)* enables each ISD to set up and maintain its own roots of trust, and also to authenticate the identities of all ASes within it. This can be effectively done by issuing and disseminating SCION Certificates which bind AS identities to public keys and which are verifiable via the corresponding roots of trust. In particular, every AS is uniquely identified in SCION by the concatenation of an ISD identifier and an AS identifier, which we are going to call *ISD-AS identifier* from now on.

The TRC serves as a root certificate. It is self-signed by a quorum of Primary ASes and defines the ISD's roots of trust, i.e., a set of root public keys which are considered to be axiomatically trusted. TRC updates must also be signed by a quorum of Primary ASes, thus ensuring that each TRC can be validated against all of its previous versions, starting from the *base TRC* which bootstrapped the update chain. Moreover, all Primary ASes in an ISD are synchronized in a way that allows them to hold a consistent view over their TRC, i.e., no two TRCs should exist with same version number and different contents for the same ISD. Lastly, there is an important distinction to make between a *valid* and an *active* TRC. A TRC which is formally correct and consistent with its previous versions is said to be valid when its *validity period* has already started but not yet ended. An *active* TRC is a valid TRC which is also either the latest TRC version or the previous one, if this is still considered to be in its *grace period*. The exact structure of a TRC, with a short description of all its fields, can be found in Section A of the Appendix.

We distinguish between two kinds of SCION Certificates: *AS Certificates* and *Issuer Certificates*. In short, AS Certificates are used by all ASes whenever they need to produce a cryptographic signature in SCION, e.g., for signing PCBs during beaconing. On the other hand, Issuer Certificates can only be owned by a special category of Primary ASes, called *Issuing ASes*, and they are used uniquely to authenticate AS Certificates in the same ISD. The exact structure of both SCION Certificates, with a short description of all their fields, can be found in Section B of the Appendix.

Key Name	Location of Public Key	Usage
Offline Root Key	TRC	Sensitive TRC update
Online Root Key	TRC	Regular TRC update
Issuing Key	TRC	Signing Issuer Certificates
Issuer Certificate Key	Issuer Certificate	Signing AS Certificates
Signing Key	AS Certificate	Signing Control Plane messages

Table 2.1: Summary of the main private keys in SCION's CP-PKI

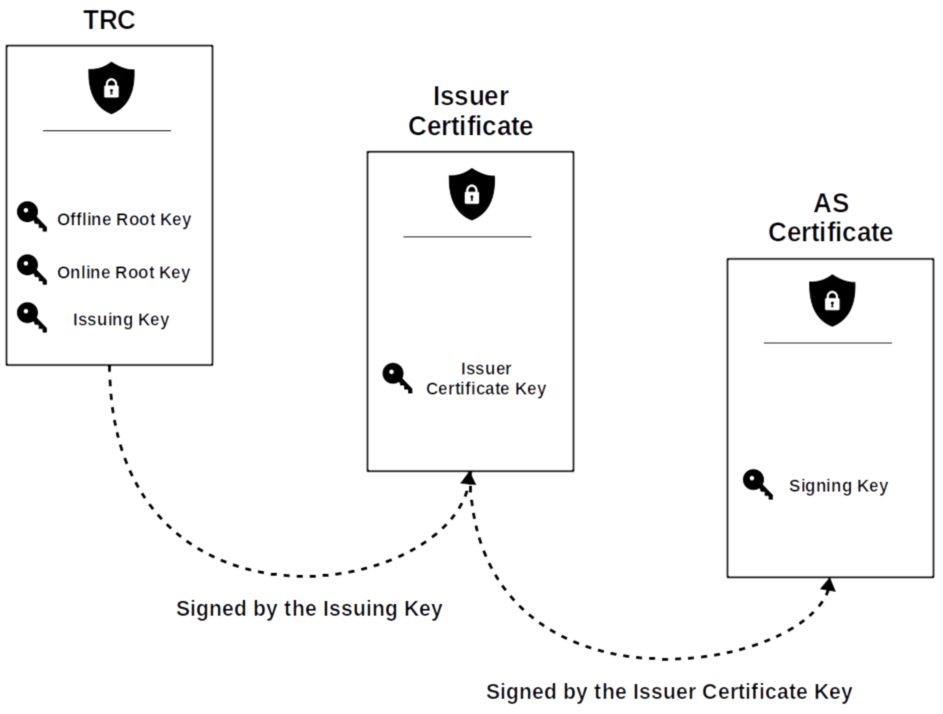

Figure 2.2: Trust flow in SCION's CP-PKI

As listed in Table 2.1, a TRC authenticates up to three public keys for each Primary AS. The first two are associated with the private keys involved in the processes of *safety-critical* and *automated TRC update*, which are, respectively, the *Offline* and the *Online Root Key*. The last one is associated with the private key responsible for the authentication of Issuer Certificates, i.e., the *Issuing Key*. To complete the picture, the authenticity of the *Signing Key* of an AS can be verified via its AS Certificate, and the authenticity of the *Issuer Certificate Key* signing this AS Certificate can be verified via the associated Issuer Certificate. In turn, the authenticity of the Issuing Key signing this Issuer Certificate follows directly from the authenticity of the TRC itself. The scheme in Figure 2.2 further illustrates how trust in SCION flows from TRCs to AS Certificates via Issuer Certificates.

Offline and Online Root Keys can only be held by a special category of Primary ASes, called *Voting ASes*. On the other hand, holding Issuing Keys and Issuer Certificate Keys is responsibility of Issuing ASes. Finally, any AS, regardless of its privileges, can hold a Signing Key.

Updating an Issuer Certificate only involves the owner Issuing AS self-signing the certificate with its own Issuing Key. By contrast, updating an AS Certificate involves both a Requesting AS and an Issuing AS signing the certificate with its own Issuer Certificate Key. We will see in detail the protocol for updating AS Certificates in Section 5.

For any given AS Certificate, there exists exactly one *Certificate Chain*. This consists of the AS Certificate itself paired with the Issuer Certificate authenticating it. Since the AS Certificate internally references this Issuer Certificate, a Certificate Chain is uniquely identified just by the pair of AS Certificate fields `(subject, version)`. See Section B of the Appendix for the full description of an AS Certificate.

Newly created or updated TRCs and Certificate Chains are made available to certificate servers across the SCION network via a process of *dissemination*. Availability of all these certificates at various entities in an AS is required, for example, when a beacon server wants to verify a PCB to propagate further, when a path server wants to verify a path segment to register, or simply, when an end-host wants to verify the authenticity of a message signed by the Signing Key of another AS. When a beacon server is missing a SCION Certificate or a TRC for verifying one of the signatures in a PCB, it can request it from the sending beacon server. When a path server is missing one of these certificates for verifying one of the signatures in a path segment, it can either request it from the beacon server who is trying to register it or to the remote path server who provided the segment. Both beacon servers and paths servers take care of submitting the retrieved TRCs and SCION Certificates at a local certificate server, so that end-hosts can directly retrieve them from there. Additionally, end-hosts will receive all missing TRCs and SCION Certificates needed for a full path verification when querying their local path server for an end-to-end path to their desired communication partner. We can then say that dissemination relies on the process of beaconing for updating beacon servers, and on the processes of path registration and path lookup for updating respectively path servers and end-hosts.

2.5 Hidden Paths Design

We have already seen in Section 2.2 how path servers maintain and distribute path segments all across the SCION network in order to allow a server located inside an AS to be reached by any other remote end-host of SCION. Hence, by default, path segments in SCION are intended to be public and retrievable without restrictions by any network entity, who is then free to assemble valid forwarding paths with them.

Hidden Path communication [17] was introduced to provide a solution for all use cases where a specific service located inside an AS should only be accessed by authorized remote end-hosts. The idea behind it is to avoid registering certain path segments at regular path servers, but effectively keeping them hidden by only sharing them out-of-band with selected authorized entities. Only these entities will then be able to construct the valid forwarding path needed for reaching the remote service. Of course, restrictions can be put in place at the application layer for preventing unauthorized entities from accessing the service. However, it is particularly beneficial for precluding the possibility of a *denial-of-service attack* to completely forbid unauthorized parties from establishing connections to the service in the first place.

2.5.1 Hidden Paths Infrastructure

In the context of Hidden Path communication, all down-segments that an AS wants to maintain hidden are registered at a dedicated *Hidden Path Server (HPS)*, instead of at a regular path server. These special servers enforce access control and only allow authorized entities to retrieve the hidden path segments needed for creating forwarding paths. On the other hand, up-segments are still registered at a local path server, but in a way that end-hosts recognize as being hidden path segments.

During the *path registration* phase, a Beacon Server (BS) must now distinguish between public path segments to register at regular path servers and hidden path segments to register at HPSes. Furthermore, hidden path segments can be registered as both up- and down-segments, as up-segments only, or as down-segments only. These decisions are guided by dedicated policies which are expressed in a configuration file local to the BS.

An illustration summarizing the communication scheme of the Hidden Paths Design can be found in Figure 2.3, on page 14.

Hidden Path Group

A group of ASes among which hidden paths are shared is what we call a *Hidden Path Group (HPG)*. A *Hidden Path Group Configuration (HPGCfg)* is the configuration file governing an HPG and defining the following fields:

- `GroupID`: unique group identifier.
- `Version`: configuration file version number.
- `Owner`: ISD-AS identifier of the AS owning the HPG.
- `Writers`: ISD-AS identifiers of all ASes in the group with permission to register hidden path segments.
- `Readers`: ISD-AS identifiers of all ASes in the group with permission to query hidden path segments.
- `Registries`: ISD-AS identifiers of all ASes containing HPSes which can be used by Writer ASes of the group to register hidden path segments.

The Owner AS creates the HPG. It is also responsible for sharing out-of-band the HPGCfg with all members of the group and for keeping it updated to its latest version.

Every AS belonging to an HPG deploys at least one HPS. When queried by an authorized local end-host for a hidden path segment, an HPS can resolve the request in one of the following two ways. If the HPS belongs to a Registry AS of the HPG, it can return the requested hidden path segment directly from its database. Otherwise, if the desired hidden path segment is not already present inside its cache, the HPS resolves the request by forwarding the query to a Registry AS of the HPG and by updating its cache with the reply.

2.5.2 Hidden Paths Server Authentication

One of the unsolved security aspects of the Hidden Paths Design concerns server authentication. In particular, for the process of hidden path segment registration to be secure, a BS needs to authenticate the HPS before submitting a hidden path segment to it. Nevertheless, no mechanism providing this kind of server authentication is part of the SCION architecture yet.

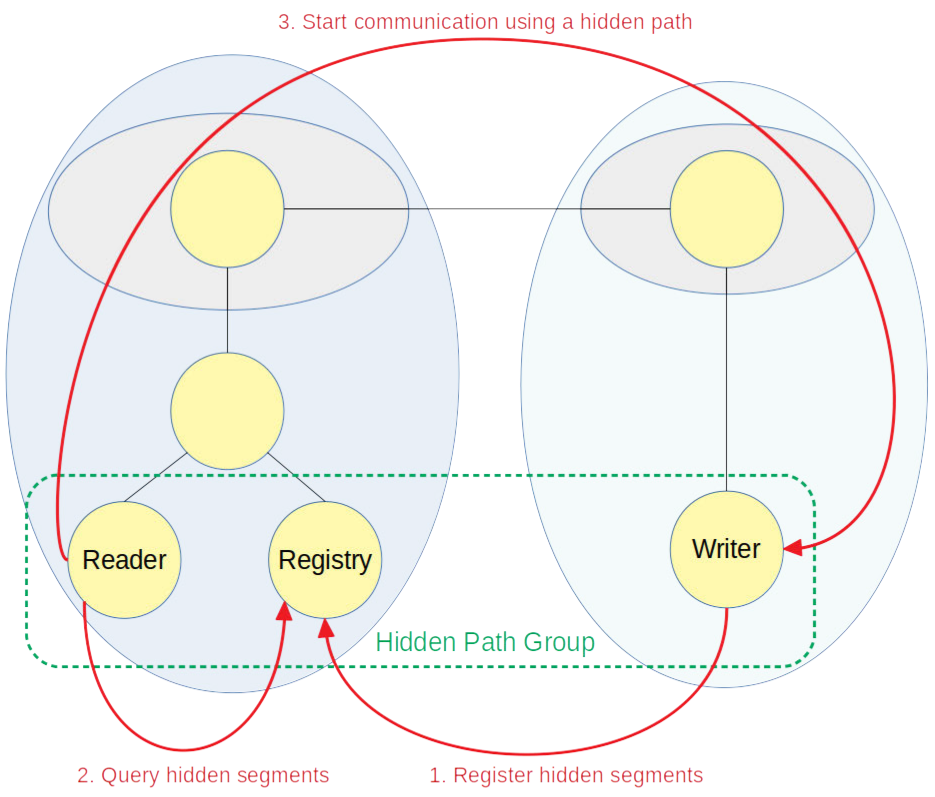
3. Start communication using a hidden path
Reader
Registry
Writer
Hidden Path Group
2. Query hidden segments
1. Register hidden segments

Figure 2.3: Hidden Path communication scheme. After an HPG is created, Writer ASes can register hidden down-segments at HPSes of Registry ASes of their group. These hidden path segments can later be queried by Reader ASes of the group to start communicating with the Writer ASes.

Chapter 3

Tamarin Prover

Given the formal specification of a security protocol and a security property defined for such a protocol, proving correctness of the protocol specification with respect to the security property is no trivial task.

The complexity of this problem largely comes from the size of the state space that needs exploration before concluding, whether the specified security property holds for the security protocol or not. In particular, the size of such state space is strongly affected by three possible sources of unboundedness in the security protocol:

1. Unbounded Messages: no restrictions on the complexity of messages that can be composed by an intruder
2. Unbounded Fresh Nonces: no limit on the number of distinct fresh nonces that can be generated in the model
3. Unbounded Sessions: no bounds on how many concurrent protocol sessions of the security protocols can be executed

In the most general case, i.e., when dealing with an unbounded number of messages, fresh nonces, and sessions, the resulting state space is infinite and the task of proving correctness of a security protocol becomes an undecidable problem. This means that no finite procedure exists for fully exploring the state space, and hence an automated verification tool such as the Tamarin prover is not guaranteed termination.

This generic unbounded setting is exactly the one we are going to consider throughout our entire formal security analysis. In particular, we are going to leverage the Tamarin prover for modeling SCION's AS Certificate Reissuance Protocol, which will be introduced in Section 5, and for proving several security properties for its specification.

In short, the Tamarin prover takes as input the symbolic model of a security protocol, the specification of the security property to be verified, and the definition of the assumed adversary capabilities. Albeit without any guarantee of termination, Tamarin's automated proof search procedure, later explained in Section 3.5, is both sound and complete. This means that, if the automated proof search algorithm terminates, the tool always returns either a formal proof of correctness or a counterexample, i.e., the trace of a possible attack violating the security property in question.

3.1 Security Protocol Model

Let us introduce the specification for a simple security protocol, which will help us explain Tamarin's modeling features throughout this entire chapter.

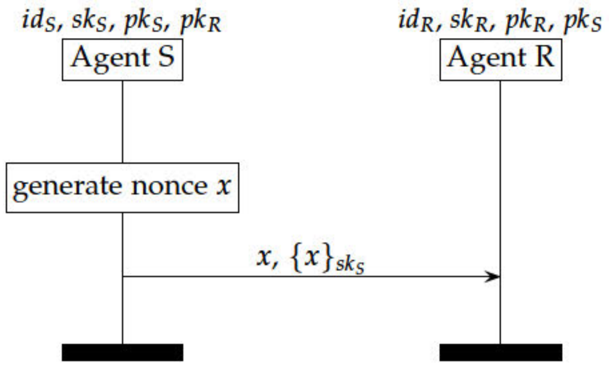
id_S, sk_S, pk_S, pk_R
Agent S
id_R, sk_R, pk_R, pk_S
Agent R
generate nonce x
x, $\{x\}_{sk_S}$

Figure 3.1: Toy Example. Agent S signs a fresh nonce and sends it to Agent R.

In Figure 3.1 we can see that our reference toy protocol is unidirectional and involves two distinct roles: a *sender* and a *receiver*. In this example, Agent S plays the role of *sender*, while Agent R plays the role of *receiver*. Each agent is characterized by its own session identifier, respectively id_S and id_R, and by a public-private key pair. Here, we have keys (sk_S, pk_S) for Agent S and (sk_R, pk_R) for Agent R. The initial knowledge of each of these agents also comprises each other's identity and public key. Agent S, instantiating the *sender* role, generates a fresh nonce x and signs it with his private key sk_S, thus producing signature $\{x\}_{sk_S}$. Then, Agent S sends both the signed nonce and its signature to Agent R, who instantiates the role of *receiver*. Agent R will only accept the received nonce after verifying the signature with respect to pk_S.

In this toy protocol, we will only consider roles having finite length. However, we will see in Section 5.4.1 that for the purposes of our analysis we will necessarily have to define infinite length roles.

The Tamarin prover enables us to reason about the correctness of this simple security protocol in an automated way. Nevertheless, we first need to model the security protocol using Tamarin's symbolic language, which is based on *multiset rewriting* and defined using three main elements: *terms*, *rules* and *facts*.

3.1.1 Terms

In Tamarin, a *term* is the most elementary building block of the modeling language. A term comes in the form of either a *variable*, a *constant*, or an n-ary *function* over terms, e.g., $f(t_1, t_2, ..., t_n)$. Tamarin already defines several built-in functions, but arbitrary function symbols can additionally be defined by the user.

We call a *substitution* a function mapping variables to terms. Moreover, we say that a term t *matches* another term q if there exists a substitution σ such that, when applied to all variables in q, equals t syntactically. This substitution σ is what we call the *matching substitution*.

The sort of a variable can optionally be made explicit using prefixes:

- ~x specifies that variable x is *fresh*
- \$$x$ specifies that variable x is *public*
- #i specifies that variable i stands for a *timepoint*

Generic terms that do not belong to any of these three sorts remain without prefix.

3.1.2 Rules

Multiset rewriting is a formalism for specifying a *labeled transition system* defined over multisets of *facts*. A multiset, or bag, is just a set where the same elements are allowed to appear multiple times. We are going to explain in detail what a *fact* is in the next subsection.

In Tamarin, a *rewrite rule* follows this pattern:

```
rule example_rule:  [ l ] -- [ a ] --> [ r ]
```

The keyword `rule` indicates the beginning of the rule named `example_rule` which describes a single transition from left-hand side l to right-hand side r, labeled as a. Letters l, r, and a are nothing but placeholders for multisets of facts. In particular, facts contained in l and r are referred to as *state facts*, because they keep track of the progress of processes in the system, while facts in a are called *action facts*, because they record the event of the rule being triggered. The ordered succession of such events is what we call *execution trace* of the transition system.

To be more explicit, a state in the transition system is a multiset containing state facts as well as special facts denoting the adversary knowledge, which we are going to introduce later in Section 3.2. Transitioning from one state to the next one is made possible via rewrite rules. A rule is only triggered when the current system state is a superset of an instantiation of its left-hand side with a matching substitution σ. When triggered, all state facts in the right-hand side of the rule are instantiated with σ and added to the next system state. On the other hand, all instantiations of the action facts of the rule are added to the execution trace along with the timepoint of the triggering event.

3.1.3 Facts

In Tamarin, a *fact* follows this pattern:

$$F(t_1, ..., t_n)$$

F is a fact symbol, while each t_i is a *term*. Arity of a fact is fixed, i.e., the same fact symbol cannot appear multiple times with a different number of arguments.

By default, a fact in Tamarin is *linear*. If a linear fact occurs on the left-hand side of a rule but not on the right-hand side, then this fact is consumed by the rule and hence removed from the next state of the transition system. On the other hand, a *persistent* fact is never removed from the state once it is introduced. In Tamarin, a fact is persistent when prefixed with a bang, e.g., `!F(t_1, ..., t_n)`.

A special fact called `Fr` is used to model the generation of unique fresh values. This fact can only appear on the left-hand side of a rewrite rule and its only argument is the fresh value being generated. Fresh values are generally used to represent random nonces, cryptographic keys, and identifiers. In Section 5.3.6 we will see how these fresh nonces can also be used to model timestamps in cryptographic certificates.

Two additional special facts are also needed to model the interaction of protocol participants with the adversary-controlled network (see Section 3.2 for details about Tamarin's threat model). The `In` fact models an agent receiving a message from the network and can only be found on the left-hand side of a rewrite rule, while the `Out` fact models an agent sending a message out to the network and can only be found on the right-hand side.

3.1.4 Security Protocol Theory

```
 1 theory ToySecurityProtocol
 2 begin
 3
 4 builtins: signing
 5
 6 // Public key infrastructure
 7 rule Register_pk:
 8   [ Fr(~skA) ]
 9   -->
10   [ !Ltk($A, ~skA), !Pk($A, pk(~skA)),  Out(pk(~skA)) ]
11
12 // Initialization rules
13 rule Init_S:
14   [ Fr(~idS), !Ltk($S, ~skS), !Pk($R, pkR) ]
15   --[ Create_S($S, ~idS) ]->
16   [ Initial_State_S($S, ~idS, ~skS, pkR, $R) ]
17
18 rule Init_R:
19   [ Fr(~idR), !Ltk($R, ~skR), !Pk($S, pkS) ]
20   --[ Create_R($R, ~idR) ]->
21   [ Initial_State_R($R, ~idR, ~skR, pkS, $S) ]
22
23
24 // Protocol rules
25 rule Agent_S_send:
26   let message = <~x, sign(~x, ~skS)>
27   in
28   [ Initial_State_S($S, ~idS, ~skS, pkR, $R), Fr(~x) ]
29   --[ Send($S, message) ]->
30   [ Out(message) ]
31
32 rule Agent_R_receive:
33   let message = <x, signed_x>
34   in
35   [ Initial_State_R($R, ~idR, ~skR, pkS, $S), In(message) ]
36   --[ Recv($R, message), Equal(verify(signed_x, x, pkS), true) ]->
37   [ ]
38
39 // Restriction
40 restriction Equal:
41   "All x y #i. Equal(x, y) @i ==> (x = y)"
42
43 end
```

Listing 3.1: Tamarin Model for the Toy Security Protocol from Figure 3.1

Listing 3.1 shows how the toy security protocol defined in Figure 3.1 can be modeled in Tamarin. The entire model is contained in a single *theory file* called `ToySecurityProtocol`. Tamarin defines several built-in *message theories*, such as `hashing`, `signing`, `asymmetric-encryption`, and `multiset`. These built-ins consist of functions and *equational theories* specifying the properties of those functions, which can be imported into the model.

In this toy example, we only need to import the `signing` built-in message theory (see line 4), which defines function symbols `sign`, `verify`, `pk`, and `true`, as well as the following equation relating all function symbols together:

```
verify(sign(m, sk), m, pk(sk)) = true
```

Only one *infrastructure rule*, namely `Register_pk`, is needed for modeling the entire PKI (see line 7). This rule generates the secret key of an agent `$A` as a fresh value and stores it into the persistent fact `!Ltk`. The corresponding public key is stored into the persistent fact `!Pk` and also sent out into the network, thus making it visible to the adversary.

Agents are initialized via *initialization rules*. Here, `Init_S` initializes an agent in the *sender* role (see line 13) and `Init_R` in the *responder* role (see line 18). These rules are responsible for setting up the agents' initial knowledge for the current session. They take as input persistent facts from the infrastructure rules to retrieve all required public and private keys and they also generate an identifier for the current session as a fresh value.

We can clearly see from Listing 3.1 that an agent's internal state in a protocol session is merely represented via state facts. Here, we only have state facts `Initial_State_S` for Agent S and `Initial_State_R` for Agent R. These facts are created on agent initialization and imbued with the respective agent's initial knowledge. Rule `Agent_S_send` (see line 25) consumes state fact `Initial_State_S` and models Agent S sending the signed nonce to Agent R. Rule `Agent_R_receive` (see line 32) consumes state fact `Initial_State_R` and models Agent R receiving nonce x and its signature. At this point, Agents S and R can run the protocol a second time only after initializing a new session, because the current session is over and the agents' states need to be recreated.

3.2 Adversary Model

The Tamarin prover works under the assumption of a worst-case adversary who controls the entire network. This active saboteur, best described in [8] and widely known in literature as the Dolev-Yao adversary, can intercept or delete any message in the network and additionally inject newly constructed messages, which can be arbitrarily derived from his knowledge by applying public functions and by generating fresh values.

Adversary knowledge is represented via the special fact `K`. For instance, `K(x)` indicates that term x is known to the adversary. Built-in rules in Tamarin make sure that all terms appearing in `Out` and `In` facts are also known to the adversary, thus modeling the assumption of a completely corrupt network. Adversary capabilities can be extended by the user via additional rewrite rules. For instance, in the setup of our running example from Listing 3.1, we can define the adversary compromising an agent in the system via the following *reveal rule*, which models an agent's private key being revealed to the network.

```
rule Reveal_ltk:
  [ !Ltk($A, ~skA) ] --[ Reveal($A) ]-> [ Out(~skA) ]
```

Listing 3.2: Reveal rule in Tamarin

3.3 Security Properties

Security properties are specified as first-order logic formulas and are checked against the execution traces of the transition system. More precisely, in Tamarin these formulas are called *lemmas* and are defined over action facts and timepoints. Existential and universal quantification is possible both over the terms contained in action facts and over timepoints, using keywords `Ex` and `All` respectively. Syntactically, Tamarin allows us to define action fact F at timepoint i as `F @i`, and to check for equality or temporal ordering between timepoints as `#i = #j` and `#i < #j` respectively.

We can also use lemmas to prove executability of the modeled protocol, hence ensuring that all security properties that we are proving do not just verify vacuously because our protocol does not run to completion. We can achieve this level of assurance by leveraging the `exists-trace` keyword. When added to a lemma, this keyword says that the lemma can be considered to be verified even if the formula only holds over one single trace. In all other cases, the formula must hold over all possible traces of the protocol for the lemma to be considered verified.

Here is a simple sanity check for our toy security protocol from Listing 3.1:

```
lemma executable:
  exists-trace
  "Ex R message #i. Recv(R, message) @i & not (Ex A #j. Reveal(A)@j)"
```

Listing 3.3: Executability check in Tamarin

3.4 Restrictions

Restrictions in Tamarin are used to reduce the number of traces of the transition system which are considered in the security analysis. Restrictions are expressed using the same first-order logic language as security properties, but prefixed with keyword `restriction` instead of `lemma`.

In our toy security protocol from Listing 3.1, the equality restriction starting at line 40 is necessary to make sure that Agent R only accepts nonce x when correctly signed by Agent S. This is because the restriction forces the protocol analysis to only consider traces where the output of the `verify` function equals the constant `true` (see line 36).

3.5 Proof Search

The Tamarin prover relies on a technique called *symbolic backwards search* to prove validity of a security property for a given protocol. The idea behind this approach is starting the proof search from the negated security property, i.e., from the symbolic representation of possible attack states. From there, trace exploration proceeds backwards trying to find if a valid initial state is reachable. If such an initial state is found, this means that the security property is violated and that the trace leading there, which describes a possible attack over the protocol, serves as a counterexample for it. Otherwise, if the proof search algorithm terminates before any valid initial state is found, the security property is said to be verified.

As previously mentioned in this chapter, this search algorithm is both sound and complete. However, since the problem of verifying correctness of a security protocol is undecideable, the algorithm is not guaranteed to terminate.

Proofs for security properties can be constructed in two different ways in Tamarin: automatically or interactively. Tamarin's *automated mode* relies on heuristics to guide the proof search without receiving any user input during the process. If the automatic proof fails to reach termination, users can resort to guiding the proof search manually with Tamarin's *interactive mode* in hopes of reaching either a proof of correctness or a counterexample for the desired security property.

Chapter 4

Problem Statement and Proposed Solution

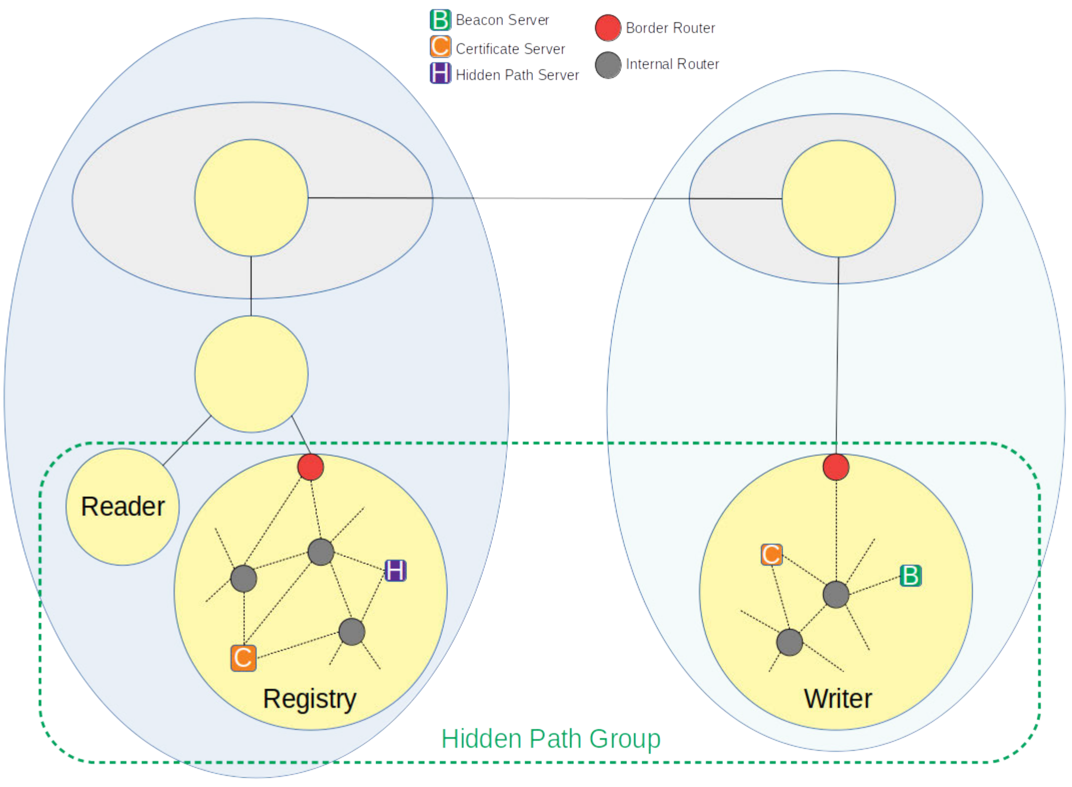

Figure 4.1: Magnified view of sections of Registry and Writer ASes from the same HPG with depiction of some of their servers and routers. This diagram refers to the same SCION topology from Figure 2.3, on page 14. For the hidden path segment registration process, it is the BS in the Writer AS who is responsible for registering hidden down-segments at the HPS in the Registry AS.

We have already mentioned in Section 2.5.2, when dealing with the security of the Hidden Paths Design, how SCION is currently lacking a mechanism for providing server authentication. Engineers at *Anapaya Systems* [1], a spin-off of ETH Zurich in charge of leading the implementation of the SCION architecture and responsible for bringing it to the market, have long been working to address this weakness. In particular, their goal is to come up with a practical solution that can be easily deployed on their implementation of the SCION architecture, currently maintained in the company's public GitHub repository[1].

In July 2019, this problem was pointed out publicly in a GitHub Issue [16]. The Anapaya Team has since explained to us that, in the context of hidden path segment registration, not only authentication of the HPS, but also authentication of the BS and confidentiality of the exchanged down-segments are desired for meeting the security goals of the Hidden Paths Design. Moreover, we have been informed by the people at Anapaya Systems that, due to other engineering constraints, they are required to exchange all Control Plane messages involved in the Hidden Paths Design [18], including hidden path segment registration packets, over encrypted QUIC [9] channels.

This technical constraint imposes some restrictions on the solution space as well. That is because QUIC is a transport protocol implemented on top of UDP capable of establishing end-to-end secure channels, i.e., authenticated at both ends and confidential, which in turn uses TLS certificates in its handshake protocol. Hence, if we trust the correctness of the QUIC protocol, the problem of setting up a secure channel between a BS and an HPS, such as in the setting depicted in Figure 4.1, boils down to finding a way for the two counterparts to exchange mutually trusted TLS certificates authenticating the asymmetric keys used for bootstrapping an end-to-end encrypted QUIC channel.

In the above-mentioned GitHub Issue, the Anapaya Team has also presented an intuition on how to solve this problem of establishing initial TLS trust, which will be introduced in Section 4.1. Their request to the Information Security group was then to come up with the design of a new protocol realizing their intuition in SCION, which we are going to introduce in Section 4.3, and to provide some assurance over its security properties by leveraging Tamarin's automated security protocol analysis. In Section 4.2 we are also going to mention the main assumptions over the SCION architecture guiding the design of our concrete solution, while in Section 4.4 we will discuss the advantages and disadvantages of the ideas behind it.

[1]`SCION` codebase: https://github.com/scionproto/scion. Accessed: 2020-02-25

4.1 Intuition

To recapitulate, in order to run the handshake needed for establishing an end-to-end secure QUIC channel between a BS and an HPS located in a different AS, we need each endpoint to share a TLS certificate that the counterpart can trust to be authentic. Nevertheless, before diving into the intuition that the Anapaya Team suggested for solving this problem of TLS trust bootstrapping, there is an important distinction that needs to made about the granularity of authentication achievable when establishing a QUIC channel between two ASes.

On the one hand, we could be interested in achieving authentication at the AS level only. This would require having a single TLS certificate per AS to be shared among all services within it. In this way, a BS can only be sure to be sending the hidden down-segments to the intended Registry AS, but not to the intended HPS within it. Similarly, the HPS can only be sure to be registering a hidden path segment coming from an acceptable Writer AS, but with no guarantees on which BS within it has actually submitted it.

On the other hand, a finer grained level of authentication would require each service in an AS to be identified by a different TLS certificate. In such a setting, the BS and the HPS are able to authenticate each other with precision at the service level. This implies that, once they have established the QUIC channel, they can be sure to be communicating with the intended counterpart in a confidential way, secure even with respect to other services located in the same source and destination ASes.

Two different, but somehow related, intuitions were then proposed by the Anapaya Team to solve the problem for both granularity levels.

In the first case, the intuition is to have each AS embed a self-signed TLS certificate into its own AS Certificate. This entails adding the self-signed TLS certificate as a new field in the AS Certificate, so that it will also be included in the signature produced by the respective Issuing AS when authenticating the AS Certificate as a whole.

Hence, this first proposal suggests having the TLS certificates from the Registry AS and the Writer AS authenticated via the following certificate trust flow:

$$\text{TRC} \rightarrow \text{Issuer Certificate} \rightarrow \text{AS Certificate} \rightarrow \text{TLS Certificate}$$

In the second case, where we are interested in distinguishing among multiple identities within the same AS, the proposal is to have a local *Certificate Authority (CA)* adding a new certification layer between the AS Certificate and the various TLS certificates for the entities in the AS. Note how the approach from the previous case does not efficiently scale to a setting with multiple distinct identities in the same AS, since this would require to embed an unspecified number of TLS certificates into the same AS Certificate, which would be inherently impractical. The proposed idea here is still to only embed one certificate into the AS Certificate, namely, a self-signed CA certificate. The purpose of this additional certificate is to authenticate all other TLS certificates in the AS, thus adding a new segment to the chain of trust needed for bootstrapping a secure QUIC channel between two ASes.

Hence, this second proposal suggests having the TLS certificates from the BS and the HPS authenticated via the following certificate trust flow:

$$\text{TRC} \rightarrow \text{Issuer Certificate} \rightarrow \text{AS Certificate} \rightarrow \text{CA Certificate} \rightarrow \text{TLS Certificate}$$

In the end, we understood that both granularity levels are achievable with the same unified solution, which is so flexible that it even allows to combine different authentication granularities at opposite ends of the QUIC channel. We will see in Section 5.2 how the AS Certificate format, exhaustively described in Section B of the Appendix, was extended to include two new fields: one for embedding a TLS certificate and the other for embedding a CA certificate. In this way, each entity across SCION can choose independently which certificate to use, and therefore which level of authentication to employ, when establishing an end-to-end secure QUIC channel to another AS.

4.2 Assumptions

Engineers at Anapaya Systems have explicitly asked us to consider the ASes as *black boxes* when coming up with a concrete solution realizing their intuitions. This means considering an AS as an entity whose inner workings are left to the AS itself to sort out independently, provided that they allow the AS to interact with other entities of the network as dictated by the protocols governing SCION.

The reason behind this request is that the solution we should provide them with is intended to become an integral part of the specification of the Hidden Paths Design, which in turn is part of the broader SCION architecture. However, SCION, due to the isolation properties that it was built upon, lacks the authority to prescribe the internal topology of an AS or the specific technologies employed for intra-AS communication. Hence, designing the internals of an AS remains an orthogonal problem to our search for a protocol for bootstrapping TLS trust from AS Certificates following the intuitions outlined in Section 4.1.

From here onwards, we can therefore rely on the following assumptions about the internals of an AS:

- Internal traffic can be secured however the AS sees fit. We can therefore assume the existence of a secure channel, i.e., authenticated and confidential, between any two entities in the same AS.
- The AS can set up a trusted local CA. This entity is assumed to operate honestly and that is why all self-signed CA and TLS certificates it produces can be fully trusted by all entities in the same AS. In virtue of the previous assumption about securing communication within an AS, we can therefore also rely on the fact that the private keys associated with these certificates can be made available in a confidential way to all the entities who are expected to know them.
- The consistency model is not part of SCION's specification. An AS may choose to deploy multiple instances of the same SCION server, e.g., more than one Certificate Server (CS), but the way that these servers are kept synchronized is left to the AS to solve. We can therefore ignore all complexities deriving from this multiplicity and just assume that an AS only deploys at most one single SCION server per typology, i.e., no more than one CS, BS, HPS, or Path Server in the same AS.

The above-mentioned assumptions allow us to think of an AS as a single irreducible node of the network when designing, and later modeling in Tamarin, the concrete protocol which realizes the intuitions presented in Section 4.1.

4.3 Realization

At this point, what is left to be done for solving the problem of bootstrapping TLS trust from AS Certificates, is to design a protocol allowing each AS to embed a local CA certificate and an AS-level TLS certificate into the current version of its own AS Certificate, as per the intuitions from Section 4.1.

Nevertheless, after analyzing all assumptions presented in Section 4.2, we concluded that there is actually no need to design a new protocol from scratch, but rather our best option is to leverage the existing mechanism of AS Certificate renewal in SCION, after extending it to suit our purposes. This mechanism is exemplified by the *AS Certificate Reissuance Protocol*, whose specification will be explained in detail in Section 5.1, and which consists of a 2-message exchange between an AS and its Issuing AS. It is worth noting that these two ASes may actually coincide. This edge case, however, does not affect the dynamics of AS Certificate renewal, since the Issuing AS can be expected to simply run the AS Certificate Reissuance Protocol with itself in such circumstances. Morover, this protocol would in practice be executed between a CS in the AS requesting the certificate reissuance and a CS in the Issuing AS responding to this request with a newly issued AS Certificate. However, given all the assumptions pointed out in Section 4.2, we can consider the parties involved in this exchange to be directly the two ASes themselves, seen as elementary nodes of the SCION network operating as black boxes, and not the two specific entities located within those ASes.

In short, in the AS Certificate Reissuance Protocol, the AS needing a renewed AS Certificate can send a reissuance request to the same Issuing AS who signed the latest version of the AS Certificate. This request should contain the proposed new version of the AS Certificate to be authenticated by the Issuing AS, together with a signature computed over it with the latest Signing Key of the AS. In turn, after checking the correctness of the request and possibly updating a few fields in the proposed certificate version, the Issuing AS will sign the new AS Certificate and reply with the corresponding Certificate Chain.

Our idea is to extend this protocol to allow the AS initiating the exchange to additionally embed a CA certificate or a TLS certificate into the proposed new version of the AS Certificate sent over to the Issuing AS. Naturally, both certificates could also be embedded together in the same request to the Issuing AS, and, in case these certificates are already present as embeddings in the latest version of the AS Certificate, they could be updated independently with each new reissuance request. The Issuing AS does not need to alter or expand the series of correctness checks performed over the incoming request before signing the new AS Certificate version. That is because, as explained in Section 4.2, the embedded self-signed CA and TLS certificates are considered to be trusted in the AS where they were created and their authenticity follows directly from the authenticity of the reissuance request where they are embedded, which, in turn, is already verified by the receiving Issuing AS by checking the signature produced with the Signing Key of the latest AS Certificate.

For the remainder of our analysis, we will focus exclusively on modeling the AS Certificate Reissuance Protocol in Tamarin. We will also try to verify several security properties for this model so as to justify the extension of the protocol that we have just proposed for solving the problem of bootstrapping TLS trust from AS Certificates. The analysis of mechanisms for establishing a trusted initial AS Certificate or for recovering after a Signing Key compromise remain instead out of the scope of our work.

4.4 Rationale for the Proposed Solution

Certainly, the decision of embedding full CA and TLS certificates into AS Certificates is not the most efficient solution for the kind of problem that we are trying to solve, given that this significantly increases the overall size of the Certificate Chains which are continuously spread across the SCION network. In fact, other approaches may have been taken for establishing a secure, i.e., authenticated and confidential, end-to-end channel between a BS and an HPS located in different ASes. However, given the technical constraints from the engineers at Anapaya Systems imposing that this channel had to be implemented via the QUIC protocol, which in turn relies on TLS certificates for authenticating the two endpoints, the solution we have presented in the course of this chapter is particularly suitable for a couple of different reasons:

1. Availability of the counterpart's embedded CA and TLS certificates, along with the entire certificate chain needed for their verification, is entirely taken care of by SCION's standard dissemination mechanisms. In particular, if a BS and an HPS located in different ASes want to communicate with each other, they first need to retrieve a verifiable end-to-end path that connects them. This can be done via a standard path lookup to a local path server, which will not only return the desired path, but also all TRCs and SCION Certificates needed for a full path verification. This necessarily includes the TRC, Issuer Certificate and AS Certificate of the destination AS, and consequently also the CA and TLS certificates which are embedded into such an AS Certificate. It is worth pointing out, instead, that whenever the BS or the HPS chooses to adopt a finer granularity of authentication, we do not need the SCION architecture to additionally take charge of distributing those TLS certificates which are directly authenticated by a CA certificate. That is because, unlike the certificate chains needed for their verification, these TLS certificates are exchanged anyway between the BS and the HPS as part of the QUIC handshake.
2. Ease of deployability. Implementing this solution as described in Section 4.3 only requires minimal changes to the SCION infrastructure. The sole preexisting protocol which actually needs to be extended is the AS Certificate Reissuance Protocol. Furthermore, this extension only affects the AS requesting the certificate renewal, who should now be given the possibility to embed a CA and a TLS certificate into the proposed new version of the AS Certificate. No change is needed on the side of the Issuing AS authenticating the reissued AS Certificate, since the additional embedded certificates are trusted and hence do not need to be checked in any way.

For the sake of completeness, let us consider a couple of alternative solutions that do not rely on the embedding of self-signed CA and TLS certificates for bootstrapping TLS trust from AS Certificates, and let us explain how these solutions are disadvantageous with respect to our specific needs.

One possibility would be to extend SCION's current Certificate Chain format to also include a CA and a TLS certificate, thus having these two additional certificates signed directly by the Issuer Certificate Key rather than simply self-signed and embedded into the AS Certificate.

Even though this solution may appear to improve the overall efficiency of SCION's Control Plane, given that it spares the AS Certificate from increasing in size, it would actually greatly complicate the distribution of CA and TLS certificates across SCION. This is because it removes the hard link between these certificates and the AS Certificate, which, in the solution we chose, is achieved via the process of embedding. The result is that SCION's already rather complex model for trust material dissemination, currently comprising a layer for TRC dissemination and a layer for Issuer Certificate and AS Certificate dissemination, would need to be extended with a new layer just for CA and TLS certificate dissemination. To complicate things even further, adding a new layer to the model for trust material dissemination would also imply adding a new layer to the model for trust material revocation. Moreover, this solution would be far more expensive to implement and deploy, since it would require several SCION's services to adapt to this extended format for Certificate Chains.

Another option would be to keep the CA and TLS certificates embedded into the AS Certificate, so as to simplify dissemination, but having them signed directly by the Signing Key of the AS rather than just self-signed.

In this case, the resulting trust flow would be equivalent to the one we get for our chosen solution. Nevertheless, this suggested solution would seriously hamper the regular process of QUIC channel establishment. This is because, when the exchanged certificates are trusted self-signed TLS certificates or TLS certificates authenticated by a trusted self-signed CA certificate, the QUIC handshake executes without any need for further inputs. However, dealing with TLS and CA certificates which are directly authenticated by AS Certificates would require the standard TLS libraries used by the QUIC protocol to be rewritten, so as to allow for the verification of these certificates based on SCION's roots of trust. This proposal would be particularly impractical to implement, since renouncing the out-of-the-box support provided by the standard TLS libraries comes with significant coding effort and security risks.

Chapter 5

Reissuance Protocol

The *AS Certificate Reissuance Protocol*, simply referred to as the *Reissuance Protocol* for the remainder of this chapter, is the mechanism that all ASes in SCION, Primary ASes included, have to use whenever they need to get their current AS Certificate renewed. A request for AS Certificate renewal may be issued, for example, when the validity period of the latest AS Certificate or of the Issuer Certificate authenticating it is about to expire, or when the AS needs to update one of the public keys contained in the AS Certificate. Refer to Section B of the Appendix for an overview of all the fields making up an AS Certificate.

This protocol, as we have previously mentioned in Section 4.3, consists of a 2-message exchange between the AS initiating the interaction, that we will call *Requesting AS*, and the *Issuing AS* responding to the request with a new AS Certificate version. For reasons already discussed in Section 4.2, we can ignore the fact that the protocol is in practice executed between the CSes of these two ASes and just reason about the message exchange at the AS level.

In the course of this chapter, we will first present the full specification of the Reissuance Protocol and our proposed extension for it, which enables the optional embedding of a CA and a TLS certificate into the renewed version of the AS Certificate (see Sections 5.1 and 5.2). Later, we will explore how we were able to abstract this protocol in a way that could be modeled in Tamarin and present the resulting specification (see Sections 5.3 and 5.4).

5.1 Reissuance Protocol Specification

In this section, we are going to present a specification for the Reissuance Protocol which was directly extracted from the implementation of the SCION architecture maintained by Anapaya Systems and available at the company's public GitHub repository[1]. We are going to refer to the fields of TRCs and SCION Certificates in the same way we do when describing the format of these certificates in Sections A and B of the Appendix.

Before diving into the specifics of the protocol, though, let us first list a series of invariants concerning TRCs and Certificate Chains. These conditions, which should always hold true to preserve the integrity of SCION's CP-PKI, are going to be helpful for understanding all the steps of the Reissuance Protocol and the consistency checks performed in it.

- Certificate Chains must be uniquely identified by the pair of AS Certificate fields `(subject, version)`, as already explained in Section 2.4.
- The owners of the two certificates in a Certificate Chain must reside in the same AS.
- The validity period of an AS Certificate must be covered by the validity period of the Issuer Certificate in the same Certificate Chain.
- The validity period of an Issuer Certificate must be covered by the validity period of the TRC authenticating it.
- An Issuer Certificate can still be considered valid and verifiable even when the TRC version authenticating it is no longer active, as long as the Issuing Key which has signed it is still associated with a newer active version of the same TRC.

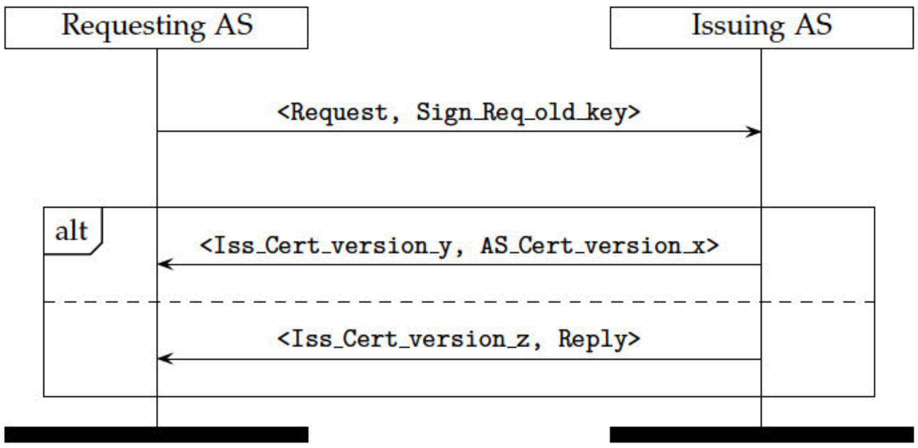
Requesting AS
Issuing AS
<Request, Sign_Req_old_key>
alt
<Iss_Cert_version_y, AS_Cert_version_x>
<Iss_Cert_version_z, Reply>

Figure 5.1: AS Certificate Reissuance Protocol Overview. Refer to Section C of the Appendix for the details about the content of the messages exchanged. The response from the Issuing AS depends on the condition given in Step 3 in the second phase of the protocol, listed in Section C.2 of the Appendix. In this message sequence chart, we use an alternative composition, labeled as *alt*, to represent this conditional response.

[1] SCION codebase: https://github.com/scionproto/scion. Accessed: 2020-02-25

In Figure 5.1, we have depicted an overview of the messages exchanged between the Requesting AS and the Issuing AS. In Sections 5.1.1 and 5.1.2 we are going to present the full specification of the Reissuance Protocol, informally. The full formal specification of the Reissuance Protocol is left to Section C of the Appendix.

5.1.1 Initial Knowledge

We have already seen that the Reissuance Protocol is executed between a Requesting AS and an Issuing AS located in the same ISD. What is more, both ASes need to hold some initial knowledge before they can initiate the protocol.

In particular, we assume that they both have access to:

- A clock synchronized with all other ASes in the ISD.
- All TRCs issued for the ISD, starting from the current base TRC which bootstrapped the update chain.

Additionally, the Issuing AS is expected to know:

- All AS Certificates issued for the Requesting AS.
- All Issuer Certificates owned by the Issuing AS and the Issuer Certificate Keys associated with them.

On the other hand, the Requesting AS is expected to know:

- A Certificate Chain where the AS Certificate contained in it is still valid and is the latest version in store owned by the Requesting AS.
- The Signing Key associated with the AS Certificate of this Certificate Chain.

5.1.2 Message Exchange

The entire Reissuance Protocol can be subdivided into three main phases, which we are going to call, in order of execution, *Requesting AS Send*, *Issuing AS Receive and Send*, and *Requesting AS Receive*. In the first phase, the Requesting AS produces a reissuance request and sends it over to the Issuing AS. In the second phase, the Issuing AS receives the reissuance request, elaborates an appropriate response, and sends it back to the Requesting AS. In the last phase, the Requesting AS receives the response from the Issuing AS and, if all correctness checks succeed, updates its own knowledge with the received SCION Certificates.

Below, we are now going to present each one of these three phases in more detail. We will do this by giving an informal explanation of all of their fundamental execution steps. Throughout the course of this description, we will refer to Section C.2 of the Appendix for the formal specification of these individual steps.

1. Requesting AS Send.

The Requesting AS makes a copy of the AS Certificate contained in the Certificate Chain that we have mentioned in Section 5.1.1 (Step 1.1). This AS Certificate version is part of the initial knowledge of the Requesting AS and serves as the base for the proposed new version of the AS Certificate to be sent over to the Issuing AS. Before it is ready, though, the version number of this copy must be incremented by one (Step 1.2). Optionally, the Requesting AS can also decide to update its own Signing Key with this reissuance request (Step 1.3). That can be done by incrementing the version number of the Signing Key by one, and by updating the corresponding public key accordingly. In addition to this, the Requesting AS can also decide to change the algorithm that the Signing Key can be used with by updating the corresponding algorithm identifier. Eventually, this modified copy of the latest AS Certificate version held in store by the Requesting AS gets signed by the Requesting AS with the possibly updated Signing Key (Steps 1.4 and 1.5). The resulting certificate is now ready to be sent over to the Issuing AS as the proposed new version of the AS Certificate.

This proposed new version of the AS Certificate, though, must be paired with a new signature computed over it by the Requesting AS with the Signing Key of the AS Certificate that was used as a base for it (Step 1.6). The two signatures contained in the resultant AS Certificate renewal request sent over to the Issuing AS (Step 1.7) serve as proofs of possession for the Signing Keys held by the Requesting AS.

2. Issuing AS Receive and Send.

The Issuing AS receives the reissuance request from the Requesting AS (Step 2.1). In order to compute all the necessary correctness checks over this incoming request, the Issuing AS retrieves a Certificate Chain where the AS Certificate contained in it is still valid and is the latest version issued for the Requesting AS (Step 2.2). Note that, if this AS Certificate version happened to have already expired, the Issuing AS would not be able to reissue any newer version of the AS Certificate for the Requesting AS and so its update chain would come to a standstill. In practice, this has to be considered an anomalous event which would require some other mechanism outside the scope of our analysis to produce and distribute out-of-band a new valid version of the AS Certificate, thus restarting the update chain.

Now, in Section 5.1.1 we have assumed that the Issuing AS, on the basis that it is a Primary AS, knows all the AS Certificates issued for the Requesting

AS. Nevertheless, the Requesting AS itself is not guaranteed to hold all of them in its store and it may even be missing the latest AS Certificate issued for it. This kind of situation, where the two ASes are out of sync and have a discording view on which AS Certificate issued for the Requesting AS is the latest, may in practice be caused by a reset of some CS in the Requesting AS or by a fault in the inter-AS SCION Certificate distribution mechanisms.

To remedy this possible inconsistency, the Issuing AS checks the version number of the proposed new version of the AS Certificate contained in the incoming request (Step 2.3). If this is less than or equal to the version number of the latest AS Certificate from the point of view of the Issuing AS, then it means that we are in the situation described above where the Requesting AS has lost some of its own AS Certificates. The Issuing AS will then update the Requesting AS by sending back the most recent Certificate Chain issued for it. On the other hand, if the version number of the latest AS Certificate from the point of view of the Issuing AS is exactly one shy of the version number of the proposed new version of the AS Certificate, then the two ASes are in sync and the Issuing AS can continue with the reissuance of the AS Certificate.

Hence, the Issuing AS proceeds with the verification of the two signatures contained in the incoming AS Certificate renewal request (Steps 2.4 and 2.6) and with the validation of the proposed new version of the AS Certificate contained in it (Step 2.5). In particular, this proposed certificate should be owned by the Requesting AS and it should indicate that it was originally authenticated by the same Issuer Certificate contained in the Certificate Chain which was retrieved by the Issuing AS at the beginning of this second phase of the protocol. Moreover, if the public key listed in this proposed new version of the AS Certificate has changed from the previous version, then the version number of the corresponding Signing Key should also have increased by one.

Next, the Issuing AS can start to assemble a new Certificate Chain (Step 2.7). The Issuer Certificate contained in it is the latest version owned by the Issuing AS and is supposed to still be valid at the time of reissuance. Necessarily, this Issuer Certificate is going to have a version number greater than or equal to the Issuer Certificate originally authenticating the proposed new version of the AS Certificate. Now, the Issuing AS proceeds with changing the information about the issuer in the proposed new version of the AS Certificate, so that it points to the Issuer Certificate contained in the new Certificate Chain. Moreover, the validity period of this proposed certificate is updated, so as to be fully covered by the validity period of the Issuer Certificate in this new chain.

Please note, however, that the validity period of the reissued AS Certificate could in practice be set either by the Requesting AS or by the Issuing AS, based on policies specific to the ISD. In this specification for the Reissuance Protocol, we arbitrarily chose to have the Issuing AS performing this operation. The advantage of this choice is that the Issuing AS can avoid checking the validity period of the incoming request, since this is going to be overwritten anyway.

Lastly, the Issuing AS signs the resulting AS Certificate with the Issuer Certificate Key associated with the Issuer Certificate in the new Certificate Chain. This AS Certificate is added to the new Certificate Chain, which is now officially issued, locally stored by the Issuing AS (Step 2.8), and finally sent back to the Requesting AS (Step 2.9).

3. Requesting AS Receive.

In this phase of the protocol, we only describe the Requesting AS handling the receiving of a newly issued Certificate Chain from the Issuing AS. The handling of the case we have seen in Step 2.3, where the Requesting AS needs resynchronization, is instead not part of the specification of the Reissuance Protocol. That is because, besides receiving back from the Issuing AS its latest Certificate Chain, another mechanism is also needed for enabling the Requesting AS to securely recover its possibly missing Signing Key. We assume this separate protocol to be executed correctly and out-of-band, and thus we are not going to provide its specification in this context.

The Requesting AS receives the newly issued Certificate Chain from the Issuing AS (Step 3.1). It then checks that all the information about the Signing Key in the AS Certificate is left unchanged with respect to the proposed version in the reissuance request (Step 3.2), and that the Issuer Certificate authenticating it is either the latest version known by the Requesting AS or a newer one (Steps 3.3 and 3.4).

Next, the Requesting AS retrieves the latest issued TRC version and checks that it is still active (Steps 3.5 and 3.6). This TRC is necessary for executing the next step of the protocol, where the received Certificate Chain is validated (Step 3.7). In particular, the owner of the Issuer Certificate contained in it should be listed as an Issuing AS in the retrieved TRC. Moreover, the version number of the TRC referenced in this Issuer Certificate should be less than or equal to the version number of this retrieved TRC, and both TRCs should list the same public keys for the Issuing AS. The Certificate Chain should also be correct from the point of view of the validity periods of the two SCION Certificates in it. This means that the validity period of the AS Certificate should be covered by the validity period of the Issuer Certificate, which, in turn should be covered by the validity period of the TRC referenced in it. Eventually, the owners indicated in the two SCION Certificates contained in the Certificate Chain are checked to be respectively the Requesting AS and the Issuing AS, and their signatures are verified using the corresponding public keys listed in the Issuer Certificate and in the retrieved TRC.

If any of the checks performed in Step 3.7 failed, the Requesting AS repeats the entire series of checks using the *Grace TRC* in place of the latest TRC retrieved in Step 3.5. The Grace TRC is the second to last issued version of the TRC, but only if this is a version which can still be proven to be active (Step 3.8).

In the end, if all the checks have succeeded, the Requesting AS locally stores the newly received Certificate Chain as its latest (Step 3.9).

5.2 Reissuance Protocol Extension

As we have explained in Section 4.1, our solution for bootstrapping TLS trust from AS Certificates forces us to extend the AS Certificate format that we have described in Section B of the Appendix.

This extension is limited to the addition of a new section that we have called `embedded_certificates` and which consists of two fields: `CA` and `TLS`. These additional fields were created, respectively, for embedding a trusted self-signed CA certificate and a trusted self-signed TLS certificate, both generated inside the AS owning the AS Certificate.

The entire section is optional and the two fields may be empty in theory. We do not define how this section is managed during the creation of the initial version of an AS Certificate, since this is out of the scope of the Reissuance Protocol. We can therefore expect to find this section missing or one of the two fields empty in any new version of the AS Certificate.

The Reissuance Protocol also needs to be extended, but the additions are minimal. More precisely, in the first phase of the protocol, the Requesting AS is allowed to perform the following operations on the proposed new version of the AS Certificate to be sent over to the Issuing AS:

- Create the `embedded_certificates` section, if missing.
- Remove the `embedded_certificates` section, if present.
- Embed a new self-signed certificate in the `CA` field, in the `TLS` field, or in both, if they are empty.
- Update or remove the self-signed certificate of either the `CA` field, the `TLS` field, or both, if they are not empty.

We have already mentioned in Section 4.4 that, since the embedded self-signed certificates are trusted, they are never checked for correctness during the Reissuance Protocol. This implies that, in the second phase of the protocol, no modification is required on the Issuing AS side, which will simply ignore the additional embedded certificates. In the third phase, though, the Requesting AS will only accept the reissued Certificate Chain if the `embedded_certificates` section is left unchanged by the Issuing AS with respect to the first phase.

5.3 Modeling Choices and Abstractions

The bulk of our work was put into modeling the Reissuance Protocol in Tamarin and proving security properties for it. We have also separately modeled our proposed extension of the protocol, as it was described in Section 5.2, to prove that it executes as expected. Naturally, several aspects of the Reissuance Protocol, and more generally of SCION's CP-PKI, needed to be adapted in order to be modeled in Tamarin and, at times, even completely abstracted away in the model.

In the following subsections we will present the most significant modeling choices we made when producing our Tamarin models and all the abstractions resulting from those choices.

5.3.1 Signatures

We have already seen in Section 3 that Tamarin is a purely symbolic verification tool. This approach implicitly assumes that the processes of signing and verifying signatures are approximated via abstract function symbols.

When modeling TRCs and SCION Certificates in Tamarin, we will therefore abstract away all indications about the signing algorithms to be used with the listed keys, since these are an implementation detail which cannot be accurately captured with Tamarin's built-in message theories.

5.3.2 Digital Certificates

All digital certificates in SCION, i.e., TRCs, Issuer Certificates, AS Certificates, CA certificates and TLS certificates, are complex objects consisting of nested attribute-value pairs.

We are going to represent these data structures in Tamarin as *multisets of pairs*. This can be done by leveraging the `multiset` built-in theory, which introduces a new symbol, `+`, serving as an associative-commutative operator. As shown in Listings 5.1 and 5.2, this solution allows for some more symbolic flexibility if compared to a solution based on *lists of pairs*.

In particular, in Listings 5.1 and 5.2 we can see an example of two occurrences of the same fact, `AS_Certificate`, containing the representation of a mock certificate. These two instances can be used to match the same certificate in Tamarin, because the variable `rest` in Listing 5.2 can be symbolically

```
AS_Certificate(
  <'attr1', 'val1'> +
  <'attr2', 'val2'> +
  <'attr3', 'val3'> +
  <'interesting_attr', 'interesting_val'> +
  <'attr5', 'val5'>
)
```

Listing 5.1: A mock certificate in Tamarin. All of its attribute-value pairs are listed explicitly.

```
AS_Certificate(<'interesting_attr', 'interesting_val'> + rest)
```

Listing 5.2: A mock certificate in Tamarin. Only one attribute-value pair is listed explicitly.

substituted for all the missing attribute-value pairs from the certificate in Listing 5.1. This second occurrence of the fact could also be used to match multiple different certificates in the same Tamarin model, provided that they all contain the attribute-value pair which was listed explicitly. The same expressiveness could not be achieved if the attribute-value pairs in the mock certificates were contained in a list, instead of in a multiset.

5.3.3 Abstracted SCION Topology

Our Tamarin model of the SCION topology is restricted to considering only one ISD. That is because both parties involved in the Reissuance Protocol, i.e., the Requesting AS and the Issuing AS, must be located in the same ISD and all ISDs in SCION behave in principle in the same way. As a consequence of this choice, instead of using ISD-AS identifiers, we can simply adopt AS identifiers to uniquely identify an AS in our model.

What is more, all mechanisms in SCION which are not strictly part of the Reissuance Protocol are left outside the scope of our analysis, e.g., the Issuer Certificate Reissuance Protocol, the protocols used for creating the initial versions of SCION Certificates, or the quorum mechanism for TRC updates. Hence, they are simply assumed to work as expected and are not modeled in full in Tamarin. In particular, we only need to consider the existence of regular ASes and Issuing ASes. All other typologies of Primary ASes can be safely abstracted away in the model.

Naturally, following our overall assumptions from Section 4.2, we will keep reasoning about ASes as black boxes. Thus, we will not model in Tamarin any detail concerning the inner workings of an AS.

5.3.4 Abstracted TRC

We have seen in Section 5.1.1 that both the Requesting AS and the Issuing AS are supposed to have access to the entire history of TRC updates in their ISD. However, since we have chosen to only model one ISD and to abstract away altogether the process of TRC update, as explained in Section 5.3.3, we will assume the existence of a unique TRC shared among all ASes from now on.

This decision entails that we need to model neither the validity period nor the version number of this unique TRC. Since this TRC is simply given and assumed to be correct, we can also abstract away all information about the voting quorum, the grace period, and the proofs of possession for the listed keys. Moreover, we only need to consider Issuing Keys, since neither Online nor Offline Keys are directly used in the Reissuance Protocol. What is left of the TRC format listed in Section A of the Appendix which is still of interest, is a series of entries mapping AS identifiers of Issuing ASes to those public keys corresponding to their Issuing Keys.

In our Tamarin model, we have decided to represent each one of these TRC entries with a separate persistent fact, as shown in Listing 5.3. We use a specific restriction to make sure that only one of these persistent facts exists for each distinct AS, thus modeling the uniqueness of the TRC.

```
!TRC(<$AS_Identifier, pk(~issuingKey)>)
```

Listing 5.3: Persistent fact modeling an abstracted TRC entry.

5.3.5 SCION Certificates Dissemination

Each new version of a SCION Certificate is stored in a separate persistent fact, as shown in Listing 5.4.

```
!Certificate(certificate)
```

Listing 5.4: Persistent fact modeling a SCION Certificate in the model. Here, the `certificate` variable could stand either for an AS Certificate or for an Issuer Certificate.

We have already seen in Section 5.1.1 that Issuing ASes are supposed to have access to all the Issuer Certificates they own, as well as to all the AS Certificates previously issued. On the other hand, a Requesting AS may be out of sync and hence not correctly updated on which version of its AS Certificate is currently the latest issued in the model.

To keep constant track of which Issuer Certificate version is currently the latest in its update chain, we add to the Tamarin execution trace a new action fact `Past_Issuer_Certificate(old_certificate)` whenever an Issuer Certificate, here `old_certificate`, is updated to a newer version. The existence of such an action fact in the execution trace can be looked up via a specific restriction for checking if an Issuer Certificate is the latest version created in the model for a given Issuing AS. In particular, the restriction shown in Listing 5.5 makes sure that rules containing action fact

`Latest_Issuer_Certificate(certificate)` are only triggered if the version of the Issuer Certificate in the variable `certificate` is currently the latest in its update chain.

```
restriction issuer_certificate_is_latest:
  "All certificate #i.
     Latest_Issuer_Certificate(certificate) @i ==>
     not(Ex #j. (#j < #i) & Past_Issuer_Certificate(certificate) @j)"
```

Listing 5.5: Restriction ensuring that an Issuer Certificate is currently the latest issued version.

In Tamarin, to model the different view that Requesting ASes and Issuing ASes may have over AS Certificates, we track separately if an AS Certificate is the latest version issued in the model or if this is just the latest version known to its owner Requesting AS.

In particular, for checking if an AS Certificate is the latest version issued in the model, we proceed in a similar way to the case of Issuer Certificates. Hence, we add to the execution trace an action fact with fact symbol `Past_AS_Certificate_global` whenever an AS Certificate is updated, and an action fact with fact symbol `Latest_AS_Certificate_global` whenever we need to make sure, before triggering a rule, that a specific AS Certificate is the latest version in the model.

On the other hand, for checking if an AS Certificate is the latest version known to its owner Requesting AS, we first need to keep track of which AS Certificates are known by each Requesting AS. This can effectively be done in Tamarin via a separate rule that adds to the execution trace an action fact `Received_AS_Certificate($AS_Identifier, certificate)` which models the AS identified by `$AS_Identifier` receiving the AS Certificate in the variable `certificate`. The same action fact is also added to the execution trace at the end of each successful run of the Reissuance Protocol and whenever the Issuing AS needs to trigger a resynchronization of the Requesting AS in the second phase of the protocol. Later, when we want to make sure that a specific AS Certificate is the latest version known to the Requesting AS before triggering a rule, we can use a restriction that checks the execution trace for the existence of the corresponding `Received_AS_Certificate` fact and for the absence of other `Received_AS_Certificate` facts for the same Requesting AS and a higher version of the AS Certificate.

At this point, it is worth making clear that we do not need to introduce any new persistent fact to model the Requesting AS and the Issuing AS storing the private keys associated with their SCION Certificates. That is because Tamarin's symbolic language allows us to derive a private key directly from the public key associated with it. Therefore, we designed our Tamarin rules so that the Issuing AS will be able to extract all its Issuer Certificate Keys directly from the Issuer Certificates authenticating them. Similarly, the Requesting AS will be able to retrieve its Signing Keys from the AS Certificates authenticating it, but only for those AS Certificate versions which were correctly registered as received by their owner.

5.3.6 Timestamps

The Tamarin prover does not allow us to represent timestamps directly in the model. Hence, we are forced to abstract away all the time indications in the Reissuance Protocol specification. This means that the validity of a SCION Certificate cannot be determined based directly on time.

We can address this limitation by expressing the notion of validity of a certificate using a fresh value. As shown in Listing 5.6, we can therefore introduce a persistent fact to record a fresh validity indicator being created along with each new SCION Certificate.

```
!Timestamp(~timestamp)
```

Listing 5.6: Validity indicator, here called `timestamp`, stored in its persistent fact upon creation.

This validity indicator, as we are going to make more clear in Section 5.3.7, is intended to replace both timestamps contained in the `validity` object of SCION Certificates. You can refer to Section B of the Appendix for an overview of all the fields making up a SCION Certificate. The underlying idea here is to make SCION Certificates implicitly valid at creation time, hence avoiding the need to model certificates turning valid, and only using the validity indicator to model expiring certificates. This simplification is justified by the fact that our model transcends the problem of clocks out of sync in SCION, as mentioned in Section 5.1.1. Under this assumption that all clocks in an ISD are synchronized, a SCION Certificate with timestamp `validity.not_before` set to a timepoint in the future would be considered malformed and immediately discarded by any agent in the system anyway. What is more, this solution for modeling the validity of SCION Certificates intrinsically captures the correct state transition from valid to expired only, hence preventing an expired certificate from being turned valid ever again.

A new rule presented in Listing 5.7 takes care of simulating the passing of time and making these validity indicators expire. In particular, the action fact `Timestamp_has_Expired(~timestamp, certificate)` produced by this rule records in the execution trace that the validity indicator `timestamp` has expired and that the corresponding SCION Certificate contained in variable `certificate` cannot be considered valid anymore. The existence of such an action fact in the execution trace can be looked up via a specific restriction whenever, before triggering a rule, checking the validity of a SCION Certificate is needed.

```
rule Timestamp_Expiration:
  let certificate =
    <'payload', <'validity', ~timestamp> + rest> + signature
  in
  [ !Timestamp(~timestamp), !Certificate(certificate) ]
  --[ Timestamp_has_Expired(~timestamp, certificate) ]->
  [ ]
```

Listing 5.7: Rule simulating the expiration of a validity indicator.

For each Certificate Chain in the model, we also ensure via a specific restriction that the Issuer Certificate contained in it cannot expire before the AS Certificate. As a direct consequence of this restriction, we can be sure that if an AS Certificate is valid at a certain point in time, then also the Issuer Certificate authenticating it will be valid.

It is worth noting that this way of modeling the validity of a SCION Certificate remains an abstraction. What we call validity indicator and often refer to as `timestamp` in Tamarin is not a representation of an actual moment in time but rather a marker expressing whether the SCION Certificate connected to it should be considered expired or not. Hence, the practice of updating a timestamp which has not yet expired to another timestamp in the future just to extend its validity period does not translate into our model at all. For this reason, we will restrain from mentioning this kind of timestamp updates in the abstracted version of the Reissuance Protocol that we will introduce in Section 5.4.

What is more, the whole idea behind the existence of a Grace TRC in SCION is simply to provide some more flexibility to the validity period of a TRC. Nevertheless, this feature is also going to be lost in our Tamarin model as a consequence of abstracting away timestamps.

5.3.7 Abstracted SCION Certificates Structure

After applying the various abstractions presented throughout Section 5.3, the resulting SCION Certificates that we ended up modeling in Tamarin appear to be considerably streamlined when compared to their full structure that we have listed in Section B of the Appendix. We still distinguish between a `payload` section and a `signature` section, but now we only model six top-level fields for the payload: `subject`, `version`, `certificate_type`, `validity`, `signing` and `issuer`.

The meaning of the `version` and `certificate_type` fields remains unvaried. On the other hand, the `subject` field, because of the abstractions in the SCION topology described in Section 5.3.3, is now a unique AS identifier. Also, this time the `validity` field directly points to a validity indicator, as it was mentioned in Section 5.3.6.

The `signing` field replaces the `keys` field and points to an object consisting of only two fields: `key` and `key_version`. This is because we only need to model one type of key for each SCION Certificate, i.e., the Issuer Certificate Key in Issuer Certificates and the Signing Key in AS Certificates. Other key types do not take part in the Reissuance Protocol and can therefore be safely abstracted away in the model. In particular, the `key` field contains the public key contained in the SCION Certificate and the `key_version` field its version number.

The `issuer` field, in AS Certificates, remains unvaried and still points to an `issuer` object. On the other hand, in Issuer Certificates, this field can be omitted altogether. This is because, for reasons explained in Section 5.3.4, we only consider a unique TRC in the model and this implies the existence of only one possible Issuing Key for each Issuing AS that could be responsible for signing its Issuer Certificates.

As proposed in Section 5.2, to these six top-level fields we can add one more field for embedding trusted self-signed TLS and CA certificates, called `embedded_certificates`. Since these embedded certificates are trusted and the self-signed signatures over them are never verified in the context of the Reissuance Protocol, they can be reduced to only two fields: `key` and `validity`. The `key` field contains the public key of the certificate, while the `validity` field contains a validity indicator like those we have introduced for the SCION Certificates.

In order to simplify pattern matching of certificates and reduce the total number of rules used to model the Reissuance Protocol in Tamarin, we represent all AS Certificates as if they always embed both a CA and a TLS certificate. Absence of either one of these certificates is simply modeled as a dummy expired certificate.

5.4 Modeled Protocol Specification

In Section 5.3 we have discussed various abstractions that we had to resort to when trying to reduce the complexity of SCION's CP-PKI to a model that could be effectively implemented in Tamarin. As for the Reissuance Protocol itself, we have already mentioned that, in order to fit our modeling choices, it needs some adaptations too. For instance, the Requesting AS and the Issuing AS now need to deal with the streamlined format for SCION Certificates that we presented in Section 5.3.7. They also need to reason about the validity of these certificates in a whole different way, as we explained in Section 5.3.6, and adapt their behavior to the context of the restricted SCION topology we assumed in Section 5.3.3.

The full formal specification of the abstracted version of the Reissuance Protocol we ended up modeling in Tamarin can be found in Section D of the Appendix.

5.4.1 Infinite Roles

In our Tamarin model, we use infrastructure rules to create an initial instantiation of the CP-PKI. This consists of a unique TRC and of arbitrary Certificate Chains for the various ASes in the ISD.

In particular, we have a rule for creating the entries of the unique TRC, one for creating the first version of an AS Certificate for an AS, and one for creating the first version of an Issuer Certificate for an Issuing AS. These three rules model the outcomes of protocols in SCION which are not strictly part of the Reissuance Protocol and hence, as explained in Section 5.3.3, are left out of our Tamarin model and security analysis. Furthermore, we have added rules for starting update chains from these initial SCION Certificates and for extending these chains incrementally with newly issued certificate versions.

These infrastructure rules are not restricted to being executed solely before the rules implementing the actual Reissuance Protocol. To the contrary, we can expect them to be triggered at all times and even to be interleaved with successful runs of the Reissuance Protocol, so as not to exclude the possibility of a subsequent creation of new SCION Certificates or an exceptional out-of-band update of existing ones. It is also thanks to these infrastructure rules that we can model the Requesting AS and the Issuing AS possibly being out of sync with respect to the AS Certificates that they share.

What is more, we do not need to explicitly model in Tamarin the initialization of the Requesting AS and the Issuing AS. That is because all information regarding their identities and their internal state at any given time is entirely deducible from their SCION Certificates stored in persistent facts and from the execution trace, as it should be clear from Section 5.3.5. A new regular AS simply comes into existence when the first version of an AS Certificate is created for its AS identifier. In addition to that, an Issuing AS will only come to be after its Issuing Key is generated and the corresponding public key is added to the unique TRC. More generally, linear state facts are only used by the Requesting AS during the execution of the Reissuance Protocol. In this context, the Requesting AS needs a state fact to preserve its temporary knowledge of the reissuance request sent over to the IssuingAS from the first phase of the protocol, where this request is created, to the third phase, where it is used to validate the incoming response.

All of this entails that the roles instantiated by the Requesting AS and by the Issuing AS are in fact infinite, in the sense that, following their initial creation, they are never consumed after a successful run of the Reissuance Protocol. That is, on the one hand, the initial knowledge of the two ASes before running the Reissuance Protocol is derived directly from the current state of SCION's CP-PKI and from the execution trace. On the other hand, all knowledge gained from the protocol execution is added back to our model of the CP-PKI and tracked in the execution trace. Therefore, the Reissuance Protocol can be executed an infinite number of times between a Requesting AS and its Issuing AS, and, with each new run of the protocol, these two ASes keep building upon what they have obtained from the previous run, without ever resetting their knowledge.

It should not come as a surprise, after reading Section 3.5, that the combination of unrestricted infrastructure rules and infinite roles for the protocol participants negatively affects the performance of Tamarin's symbolic backwards search, since this necessarily causes an explosion of the possible states that need to be explored. We will introduce a few optimizations for our Tamarin model that address this problem when presenting the results of our security analysis in Section 6.3.

5.4.2 Model Splitting

As a way to reduce the complexity of our Tamarin model, we decided to split the Reissuance Protocol into two possible scenarios and analyze each scenario in a separate theory file. This choice turned out to be particularly helpful for improving the efficiency of the Tamarin prover when trying to verify the security properties that we are going to introduce in Section 6.2.

In particular, in the first scenario we focused on modeling the resynchronization capabilities of the Reissuance Protocol. Hence, in this theory file we implemented the first phase of the protocol in full, but we cut short the role of the Issuing AS at Step 3 in the second phase. Therefore, for all cases where the Requesting AS is missing the latest AS Certificate issued for it, we modeled the Issuing AS interrupting the Reissuance Protocol execution and triggering the resynchronization process. As explained in Section 5.3.5, this entails in Tamarin that the Issuing AS updates out-of-band the Requesting AS with the missing Certificate Chain and that the Requesting AS manages to successfully recover its possibly lost Signing Key.

On the other hand, in the second scenario we make sure that, before initiating the Reissuance Protocol, the Requesting AS and the Issuing AS are already synchronized with respect to the latest Certificate Chain that they share. Therefore, in this theory file we were able to model and analyze the full message exchange between Requesting AS and Issuing AS, comprising all three phases of the Reissuance Protocol.

Chapter 6

Proof Summary

In the course of this chapter, we will present how we were able to leverage the Tamarin prover to provide some assurance over the security properties of the modeled Reissuance Protocol specification which we have described in Section 5.4. Our goal is to formally verify those security requirements necessary to justify the extension of the Reissuance Protocol that we have specified in Section 5.2 and that we have also separately modeled in Tamarin.

In this process, we will first need to define a suitable adversary model (see Section 6.1) and specify the desired security properties to be verified (see Section 6.2). Lastly, we will explain how we were able to obtain the formal proofs of correctness for these properties, and we will present our final results (see Section 6.3).

6.1 Adversary Model

We have already seen in Section 3.2 how Tamarin's threat model assumes the existence of a Dolev-Yao adversary who fully controls network communication. There, we have also mentioned that the adversary capabilities can be extended via additional reveal rules, which model agents of the system being compromised. More specifically, we consider an agent to be compromised if at least one of its private keys was revealed to the adversary-controlled network.

To increase the flexibility of our extended adversary capabilities, we set up one distinct reveal rule for each type of key in the model. In particular, we want to give the adversary-controlled network the opportunity to receive:

- Any private key authenticated by a TLS certificate.
- Any private key authenticated by a CA certificate.
- Any Signing Key authenticated by an AS Certificate.

- Any Issuer Certificate Key authenticated by an Issuer Certificate.
- Any Issuing Key listed in the unique TRC.

In all these rules, we use action fact `Reveal($AS)` to record in the execution trace that the AS with identifier `$AS` was compromised. In Listing 6.1, we can see a reveal rule modeling Issuer Certificate Key `ltkIssuer` being sent out to the adversary-controlled network, thus leading to the compromise of the Issuing AS with identifier `$IssuingAS`.

```
rule Reveal_IssuerCertificateKey:
  [ !Certificate(
      <'payload',
        <'subject', $IssuingAS> +
        <'certificate_type', 'issuer'> +
        <'signing',
          <'key', pk(~ltkIssuer)> +
          <'key_version', keyVersion>
        > +
        rest
      > +
      signature
    )
  ]
  --[ Reveal($IssuingAS) ]->
  [ Out(~ltkIssuer) ]
```

Listing 6.1: Reveal rule for the Issuer Certificate Key authenticated by an Issuer Certificate.

6.2 Security Properties

In Tamarin, as we have seen in Section 3.3, we use lemmas to define the security properties we are interested in verifying for the model of a protocol. Throughout this section, we are going to introduce the various lemmas that we designed for proving correctness of the Reissuance Protocol.

First of all, we specified a series of consistency checks making sure that the infrastructure rules mentioned in Section 5.4.1 work as expected and are consistent with SCION's CP-PKI invariants. Besides, we leveraged the `exists-trace` keyword to define a number of executability checks ensuring that the modeled protocol can run to completion without adversary intervention in a selection of possible execution scenarios.

The first two security properties that we introduce, for simplicity referred to as *lemma 1* and *lemma 2*, are needed to verify that all the update chains of SCION Certificates follow an ordered progression, without any skipped or duplicate version number. The next two security properties, which we call *lemma 3* and *lemma 4*, take care of proving the correct advancement of the version number of the Signing Key in an AS Certificate, and, analogously, of the version number of the Issuer Certificate Key in an Issuer Certificate. These version numbers are incremented by exactly one in any newer version of the certificate where the key has changed, but remain unaltered in all other update events.

One more security property, that we simply refer to as *lemma 5*, ensures that, if a Requesting AS accepts an incoming Certificate Chain in the third phase of the Reissuance Protocol, it means that the AS Certificate contained in it was reissued by the correct Issuing AS in the second phase of the protocol after receiving the corresponding reissuance request sent by the same Requesting AS in the first phase. This security property assumes that the Requesting AS and the Issuing AS involved in the execution of the Reissuance Protocol are not compromised or otherwise the lemma would be trivially violated by the adversary impersonating either the Requesting AS in the first phase of the protocol, or the Issuing AS in the second phase. Lemma 5 is particularly useful because it will be reused by the Tamarin prover to help verifying the agreement properties that we are just about to present in Section 6.2.1.

6.2.1 Agreement Properties

Since the Reissuance Protocol only deals with public SCION Certificates and no confidential information is exchanged between the Requesting AS and the Issuing AS, we do not need to define any secrecy property for our model. We will instead focus on agreement properties, namely *identity agreement*, *injective agreement* and *strong session agreement*. For the formal definitions of these three properties, we will rely on the hierarchy of authentication specifications identified by Lowe in [11].

Before we begin to present the three agreement properties in detail, it is important to remember that, as explained in Section 5.4.2, we decided to split the Reissuance Protocol into two possible scenarios, which we have modeled in separate theory files. In the first theory file we only study the resynchronization capabilities of the Reissuance Protocol, whereas the model of the full message exchange between the Requesting AS and the Issuing AS is left to the second theory file. Now, when it comes to our agreement properties, we are actually only interested in proving them for this second scenario. That is because the resynchronization process, which, as we have seen in Section 5.1.2, is assumed to always execute correctly and out-of-band, has no undesired side effects anyway and can be expected to be triggered at any time by other mechanisms in the ISD unrelated to the Reissuance Protocol. Therefore, no agreement with the Issuing AS is required for the outcomes of a resynchronization of the Requesting AS to be consistent with the current instantiation of the CP-PKI.

Moreover, since all of our agreement properties will be defined over the messages exchanged as a whole and not over specific elements of these messages, we can restrict ourselves to only proving them for the Reissuance Protocol and avoid repeating the same proofs for the extended version of the protocol presented in Section 5.2. That is because, as we have already explained, the Reissuance Protocol specification remains substantially unvaried and no new checks are introduced on the side of the Issuing AS to deal with the extended version of the AS Certificate presented in Section 5.3.7. Hence, the TLS and CA certificates which are possibly contained in the reissuance request and in the returned Certificate Chain will simply inherit from the AS Certificates where they are embedded all the properties that we were able to prove for them in the Reissuance Protocol. Therefore, for this extended version of the Reissuance Protocol, we will limit ourselves to verifying consistency checks, executability checks, and the first four lemmas about correctness of SCION Certificate updates, so as to prove that our extended model works as expected.

Identity Agreement

With identity agreement, we are interested in making sure that whenever the Requesting AS and the Issuing AS accept a message from their counterpart, they agree on the identity of this agent. When verifying this property, as well as the other two agreement properties, we expect the Requesting AS and the Issuing AS to be honest, i.e., not compromised. Otherwise, much like in the case of lemma 5, if either one of these agents is compromised, agreement would be trivially broken.

In particular, our interpretation of identity agreement perfectly fits Lowe's formal definition of *weak agreement*, which we quote verbatim below:

Definition 6.1 (Weak Agreement) [11] *We say that a protocol guarantees to an initiator* A weak agreement *with another agent* B *if, whenever* A *(acting as initiator) completes a run of the protocol, apparently with responder* B*, then* B *has previously been running the protocol, apparently with* A*.*

In Tamarin, we express agreement properties in terms of specific action facts called *claim events* that we add as labels to the protocol rules. Here, we distinguish between two claim events: a *running claim*, with fact symbol `Running`, and a *commit claim*, with fact symbol `Commit`. In Figure 6.1, we illustrate how these claim events are laid out in the execution trace of the Reissuance Protocol for verifying weak agreement.

Furthermore, when labeling our rules in Tamarin, each commit claim must be accompanied by all the relevant *honesty claims*, i.e., one separate action fact `Honest($AS)` for each AS with identifier `$AS` which is expected to be honest. In our model, for all three of the agreement properties that we are interested in proving, these honesty claims are `Honest($R)` and `Honest($I)`, where `$R` and `$I` are the AS identifiers, respectively, of the Requesting AS and of the Issuing AS.

In Listing 6.2, we define a lemma to look up all these claims, so as to verify weak agreement for the Reissuance Protocol.

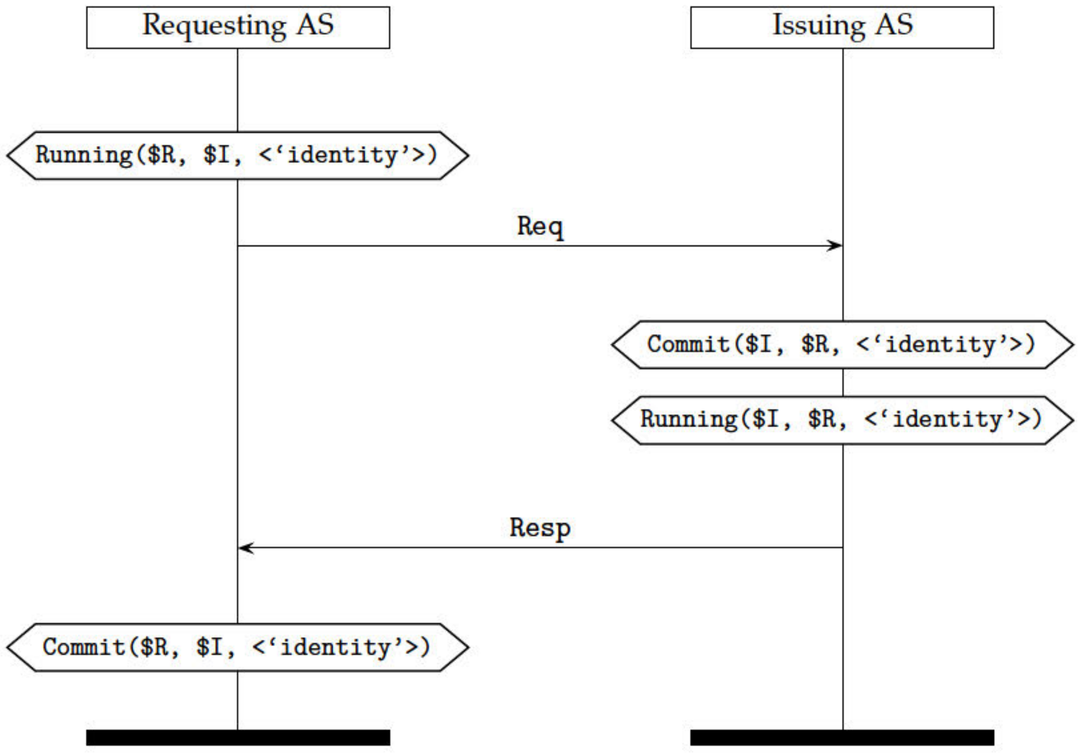
Requesting AS
Issuing AS
Running($R, $I, <'identity'>)
Req
Commit($I, $R, <'identity'>)
Running($I, $R, <'identity'>)
Resp
Commit($R, $I, <'identity'>)

Figure 6.1: Weak Agreement in the Reissuance Protocol. `$R` and `$I` are the AS identifiers, respectively, of the Requesting AS and of the Issuing AS. `Req` and `Resp` are placeholders, respectively, for the reissuance request and for the returned Certificate Chain. The honesty claims `Honest($R)` and `Honest($I)` which accompany both commit claims are omitted from the chart.

```
lemma weakagreement:
  "All a b #i.
    Commit(a,b,<'identity'>) @i
    ==> (Ex #j. Running(b,a,<'identity'>) @j)
        | (Ex X #r. Reveal(X)@r & Honest(X) @i)"
```

Listing 6.2: Lemma defining the weak agreement property.

Injective Agreement

A stronger agreement property, and actually the strongest in the hierarchy defined by Lowe, is *injective agreement*, also sometimes referred to simply as *agreement*. Its formal definition is quoted verbatim below:

Definition 6.2 (Injective Agreement) [11] *We say that a protocol guarantees to an initiator* A agreement *with a responder* B *on a set of data items* ds *if, whenever* A *(acting as initiator) completes a run of the protocol, apparently with responder* B*, then* B *has previously been running the protocol, apparently with* A*, and* B *was acting as responder in his run, and the two agents agreed on the data values corresponding to all the variables in* ds*, and each such run of* A *corresponds to a* unique *run of* B*.*

As we can derive from its definition, unlike weak agreement, injective agreement does not only ensure that the two agents agree on each other's identities, but also that they agree on their respective roles and on a set of data items exchanged during the protocol run. What is more, this property enforces a unique correspondence between matching protocol runs of the two agents, thus effectively ruling out the possibility of replay attacks. In the context of the Reissuance Protocol, these two agents are always the Requesting AS and the Issuing AS, and their roles are represented, respectively, with constants `'R'` and `'I'`. Moreover, as we can see in Figure 6.2, we want the two ASes to agree on the entire content of the messages exchanged, i.e., on the whole reissuance request and on the whole returned Certificate Chain.

In Listing 6.3, we define the lemma specifying injective agreement of the Requesting AS with the Issuing AS on the returned Certificate Chain, i.e. agreement with the responder. On the other hand, in Listing 6.4, we define the lemma specifying injective agreement of the Issuing AS with the Requesting AS on the reissuance request, i.e. agreement with the initiator.

```
lemma injectiveagreementRESPONDER:
  "All a b t #i.
    Commit(a,b,<'R','I',t>) @i
    ==> (Ex #j. Running(b,a,<'R','I',t>) @j
          & not (Ex a2 b2 #i2. Commit(a2,b2,<'R','I',t>) @i2
          & not (#i2 = #i))
        )
        | (Ex X #r. Reveal(X)@r & Honest(X) @i)"
```

Listing 6.3: Injective agreement property from the perspective of the Requesting AS.

```
lemma injectiveagreementINITIATOR:
  "All a b t #i.
    Commit(a,b,<'I','R',t>) @i
    ==> (Ex #j. Running(b,a,<'I','R',t>) @j
          & not (Ex a2 b2 #i2. Commit(a2,b2,<'I','R',t>) @i2
          & not (#i2 = #i))
        )
        | (Ex X #r. Reveal(X)@r & Honest(X) @i)"
```

Listing 6.4: Injective agreement property from the perspective of the Issuing AS.

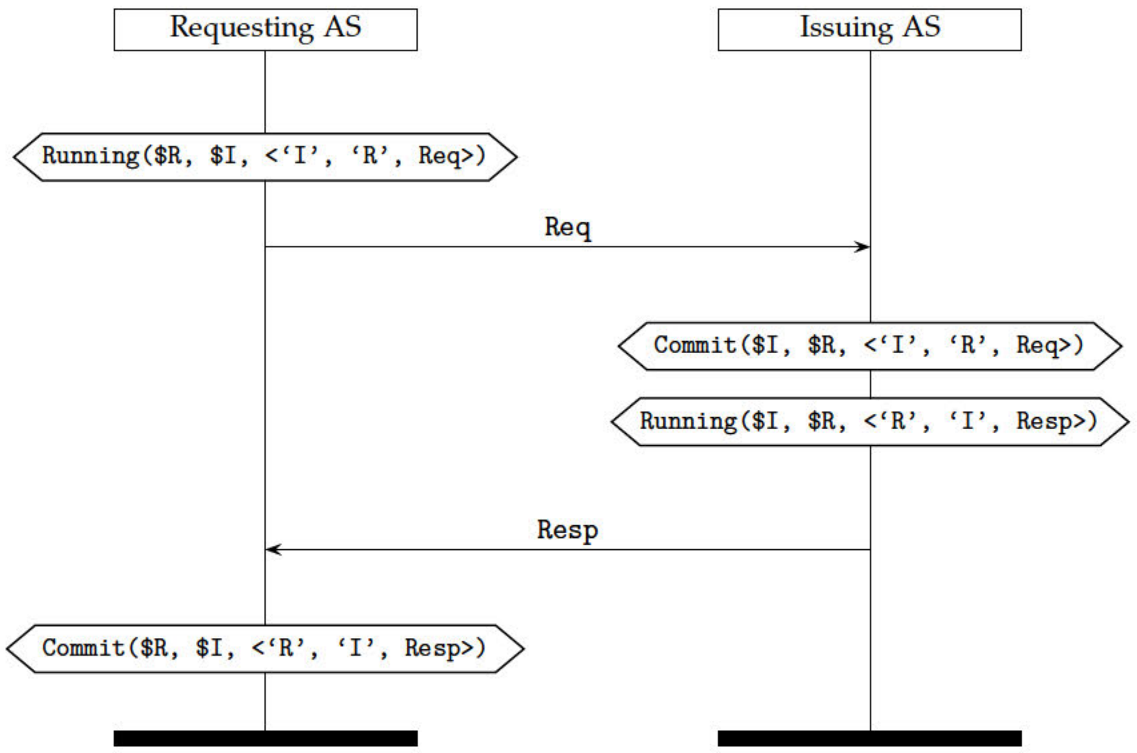
Requesting AS
Issuing AS
Running($R, $I, <'I', 'R', Req>)
Req
Commit($I, $R, <'I', 'R', Req>)
Running($I, $R, <'R', 'I', Resp>)
Resp
Commit($R, $I, <'R', 'I', Resp>)

Figure 6.2: Injective Agreement in the Reissuance Protocol. `$R` and `$I` are the AS identifiers, respectively, of the Requesting AS and of the Issuing AS. `Req` and `Resp` are placeholders, respectively, for the reissuance request and for the returned Certificate Chain. The honesty claims `Honest($R)` and `Honest($I)` which accompany both commit claims are omitted from the chart.

Strong Session Agreement

Building upon the definition of injective agreement, we can go one step further and introduce an even stronger agreement property, called *strong session agreement*. With injective agreement, as it should be clear from the chart in Figure 6.2, we can be sure that the two agents agree on each other's identities and roles, and on the two messages exchanged, when these messages are taken individually. Nevertheless, there is no way for us to be certain that the reissuance request and the returned Certificate Chain are generated and exchanged in the same protocol session. Now, with strong session agreement, we can achieve precisely this level of certainty by ensuring that the two agents agree on the full session transcript. Evidently, this property can only be verified from the point of view of the Requesting AS, since this agent is the receiver of the last message in the protocol and hence the only one who can agree with the Issuing AS on the full transcript.

The corresponding lemma can be found in Listing 6.5. In Figure 6.3, we illustrate how the claim events looked up by this lemma are laid out in the execution trace of the Reissuance Protocol for verifying strong session agreement.

```
lemma strong_session_agreement:
  "All a b t #i.
    Commit(a,b,<'session',t>) @i
    ==> (Ex #j. Running(b,a,<'session',t>) @j
          & not (Ex a2 b2 #i2. Commit(a2,b2,<'session',t>) @i2
          & not (#i2 = #i))
        )
        | (Ex X #r. Reveal(X)@r & Honest(X) @i)"
```

Listing 6.5: Lemma defining the strong session agreement property.

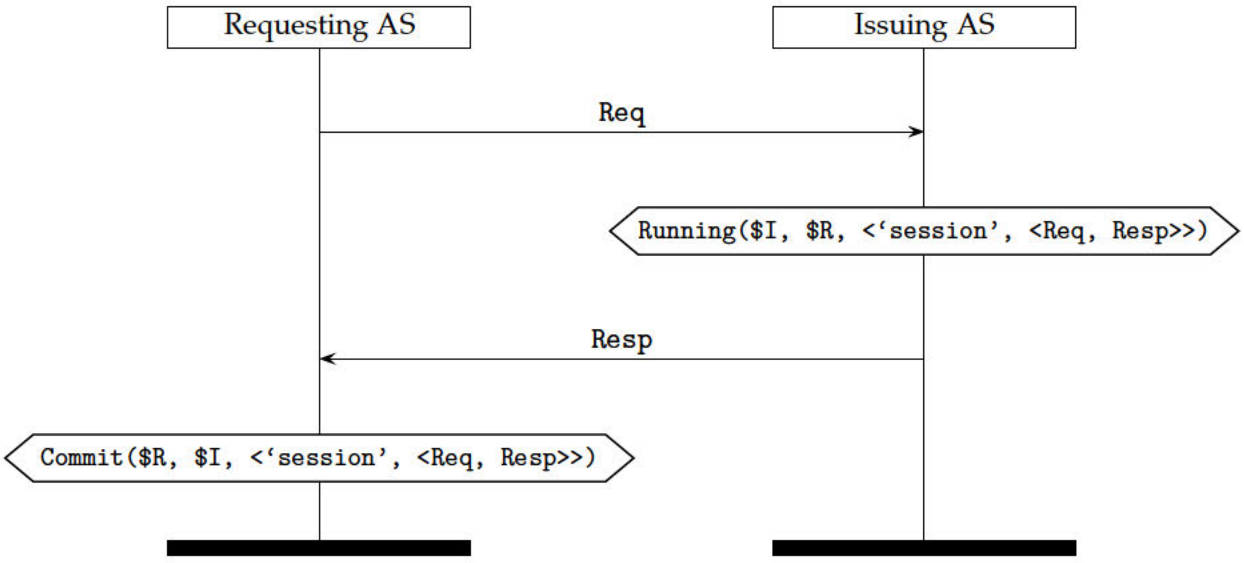
Requesting AS
Issuing AS
Req
Running($I, $R, <'session', <Req, Resp>>)
Resp
Commit($R, $I, <'session', <Req, Resp>>)

Figure 6.3: Strong Session Agreement in the Reissuance Protocol. `$R` and `$I` are the AS identifiers, respectively, of the Requesting AS and of the Issuing AS. `Req` and `Resp` are placeholders, respectively, for the reissuance request and for the returned Certificate Chain. The honesty claims `Honest($R)` and `Honest($I)` which accompany the commit claim are omitted from the chart.

6.3 Proofs and Results

All the security properties presented in Section 6.2 have been successfully verified. With the resulting formal proofs of correctness, we have now achieved the level of assurance desired to conclude that the extension of the Reissuance Protocol specified in Section 5.2 is suitable for solving our underlying problem of bootstrapping TLS trust from AS Certificates.

Most of these security properties were proven automatically by the Tamarin prover, without any need for user input during the proof search procedure. However, some agreement properties needed their proof searches to be guided manually in Tamarin's interactive mode in order for them to reach termination. More generally, the extensiveness of the Reissuance Protocol model that we have implemented in Tamarin tends to weight negatively on the number of steps necessary to reach termination and on the overall proof times. On the one hand, as we have already mentioned in Section 5.4.1, that is because of the unrestricted infrastructure rules and the infinite roles for the protocol participants which are needed for expressing the complexity of SCION's CP-PKI. On the other hand, this is also due to the fact that all messages exchanged in the protocol are made up of large multisets of pairs containing a number of different fresh values, which are on their own burdensome to deal with in Tamarin.

To make proofs shorter and easier to prove automatically, we had to devise a few expedients that helped us make our Tamarin model more efficient. In the next subsections we are going to present our main findings leading to these optimizations.

Lemma 5 Breakdown

As we have already pointed out in Section 6.2, the security property that we refer to as lemma 5 is preliminary to the verification of the agreement properties presented in Section 6.2.1. That is because the first-order formula constituting this lemma is reused directly by the Tamarin prover to simplify the proof search of the following lemmas about agreement.

Lemma 5 proved to be particularly challenging to verify. Ultimately, we decided to break it down into three simpler sublemmas so that these could be reused by the Tamarin prover to find a shorter proof for the main lemma.

- *Lemma 5.1.* This sublemma ensures that, if a Requesting AS accepts an incoming Certificate Chain in the third phase of the Reissuance Protocol, then the same Requesting AS has previously sent the corresponding reissuance request for it in the first phase of the protocol.
- *Lemma 5.2.* This sublemma ensures that, if an Issuing AS reissues an AS Certificate for a Requesting AS in the second phase of the Reissuance Protocol, then this Requesting AS has previously sent the corresponding reissuance request for it in the first phase of the protocol.
- *Lemma 5.3.* This sublemma ensures that, if a Requesting AS accepts an incoming Certificate Chain in the third phase of the Reissuance Protocol, then the AS Certificate contained in it was reissued by the correct Issuing Certificate in the second phase of the protocol.

Origin of fresh keys

To prove a security property, as we have already explained in Section 3.5, the Tamarin prover starts from its negated formula and proceeds backwards in search of a valid initial state. During this backwards search, the Tamarin prover tends to trace the origin of fresh values back to the respective instances of the `Fr` fact which generated them before deciding whether it can safely stop exploring a certain execution trace or not. This conservative behavior contributes to the strong soundness and completeness guarantees characterizing Tamarin's symbolic backwards search. Nevertheless, this feature also prolongs the proof search in all cases where the trace under analysis could intuitively be discarded earlier, regardless of any assumptions about the origin of the fresh values in it. Hence, on average, this behavior increases the number of steps that the Tamarin prover needs to execute before abandoning the exploration of an invalid execution trace.

At first glance, this may not seem like a substantial problem. However, let us not forget that our model allows for the creation of arbitrarily long update chains of SCION Certificates. Also, let us consider the fact that the number of states that need exploration is blown out of proportion by the inconvenient combination of unrestricted infrastructure rules and infinite roles for the protocol participants that we have discussed in Section 5.4.1. In light of these considerations, it should now be clear how a systematic delay in discarding invalid execution traces can make a difference when it comes to the proof search reaching termination.

When trying to address this problem, our main concern is finding a way for the public key of an AS Certificate to be directly linked back to the rule where it was first generated via the `Fr` fact, which could be either an infrastructure rule or a rule modeling the Requesting AS executing the first phase of the Reissuance Protocol. That is because, otherwise, the Tamarin prover would be forced to explore the update chain of the AS Certificate up to the certificate version where the current Signing Key was first generated. This process may require Tamarin's backwards search to go through the execution of several runs of the Reissuance Protocol, which, in turn, may involve a number of updates of the signing Issuer Certificate Key as well.

It is specifically for this purpose that we introduced the new persistent fact shown in Listing 6.6, which registers when a new public key `pk(~ltkAS)` is generated for the AS identified by `$AS_Identifier`.

```
!FreshKey($AS_Identifier, pk(~ltkAS))
```

Listing 6.6: Persistent fact registering a new Signing Key being created.

What is more, all the rules in our model where a new version of the Signing Key is created take as input both the persistent fact storing the AS Certificate to be updated and the Fr fact generating the new Signing Key. Given any one of these rules, the public key contained in the preexisting AS Certificate and the public key freshly produced in the rule itself via the Fr fact are inevitably different. That is because, as we have already mentioned in Section 3.1.3, all cryptographic keys in Tamarin are represented as fresh values and no two fresh values generated by the Fr fact are ever the same. In our particular case, one fresh value is generated in the rule itself, while the other one already existed prior to the rule being triggered. This entails that the two keys come from distinct instances of the Fr fact and must therefore be different. Nevertheless, the Tamarin prover still assumes that these two values may in principle be the same and first needs to trace back the exact rule originating the public key contained in the AS Certificate before concluding that they are in fact different. To avoid this kind of unnecessary trace exploration, we decided to use the following inequality restriction to make sure that the two public keys are always considered to be different straight away in all the rules where they appear together.

```
restriction Inequal:
  "All x y #i. Inequal(x, y) @i ==> not(x = y)"
```

Listing 6.7: Inequality restriction.

Placeholder for validity indicators

As we have explained in Section 5.3.6, we make use of specific fresh values that we call validity indicators to express whether a SCION Certificate should be considered expired or not. Moreover, we have seen in Section 5.1.2 that in our Reissuance Protocol specification it is the Issuing AS who is responsible for setting the validity period of the reissued AS Certificate. Therefore, in our Tamarin model, even though the proposed new version of the AS Certificate sent over by the Requesting AS as part of the reissuance request still contains a validity indicator that was copied over from the latest version of the AS Certificate, this validity indicator will always be overwritten by the Issuing AS without ever being checked.

To prevent the Tamarin prover from uselessly tracing back the origin of this fresh value, we simply substitute it for a unique placeholder which can be retrieved from the persistent fact with fact symbol PF. This placeholder is generated by the rule in Listing 6.8 and its uniqueness is guaranteed by the restriction in Listing 6.9.

```
rule Placeholder_Fresh:
  [ Fr(~p) ]
  --[ Placeholder_Fresh_Triggered(~p) ]->
  [ !PF(~p) ]
```

Listing 6.8: Rule creating the placeholder for validity indicators.

```
restriction Unique_Placeholder_Fresh:
  "All p1 p2 #i #j.
    Placeholder_Fresh_Triggered(p1) @i &
    Placeholder_Fresh_Triggered(p2) @j ==> (#i = #j)"
```

Listing 6.9: Restriction enforcing the uniqueness of the placeholder for validity indicators.

It is worth making clear that the solution we have just presented is not the only possible optimization when it comes to avoiding to deal with the origin of the residual validity indicator in the reissuance request. For instance, instead of using a placeholder for this fresh value, we could simply replace it with a public or a constant value. Nevertheless, this solution would require the Issuing AS to pattern match the reissuance request from the Requesting AS differently, so as to account for the new value sort. At this point, we could even omit the pair containing this validity indicator altogether from the multiset representing the proposed new version of the AS Certificate sent over to the Issuing AS. Still, we would need to alter the rule modeling the Issuing AS executing the second phase of the Reissuance Protocol to accept a shorter format for the reissuance request.

The advantage of our solution is that it preserves the sort of the validity indicator and does not require any changes in the way that the Issuing AS pattern matches the incoming reissuance request. This is important because, as we have mentioned in Section 5.1.2, the decision to always have the Issuing AS setting the validity period for the reissued AS Certificate in our Reissuance Protocol specification is completely arbitrary. In practice, some ISDs may have the Requesting AS perform this operation instead. Therefore, our chosen solution allows the Tamarin model to be easily adapted to alternative specifications of the Reissuance Protocol without any need to adjust the way that the Issuing AS receives the reissuance request from the Requesting AS, once all the references to the unique persistent fact with fact symbol PF are removed.

Chapter 7

Conclusion

This thesis has stemmed from an intuition on how to address a lack of server authentication in SCION's Hidden Paths Design. From there, we have devised a sensible solution to this issue that could easily be deployed and best fit SCION's architectural principles. We have later focused on performing a security analysis of SCION's Reissuance Protocol which was entirely supported by the Tamarin security protocol verification tool. We have leveraged this automated prover to formally model the Reissuance Protocol and to verify for it all the security properties we needed to justify our proposed solution to the authentication problem in the Hidden Paths Design, which we have also modeled in the tool as an extension of the Reissuance Protocol.

In short, it was first brought to our attention that, in the context of hidden path segment registration, a Beacon Server (BS) and a Hidden Path Server (HPS) located in different Autonomous Systems (ASes) necessitated a way to authenticate each other before sharing confidential information, but no mechanism existed in SCION for this purpose. Hence, the idea to establish an end-to-end encrypted QUIC channel between the two in order to ensure secure communication, i.e., authenticated at both ends and confidential. The problem therefore shifted to finding a way for the two servers to exchange the mutually trusted TLS certificates needed to bootstrap this encrypted QUIC channel. The crucial intuition was then to embed either a TLS certificate or a CA certificate authenticating multiple TLS certificates directly into the AS Certificates of the two ASes where the BS and the HPS are located, so as to facilitate their distribution across the SCION network.

Our main contributions have been devising a solution to realize this intuition as an extension of the Reissuance Protocol and conducting a formal security analysis to verify its legitimacy. In particular, we have decided to rely on the already existing Reissuance Protocol for securely embedding CA and TLS certificates into AS Certificates because we wanted to maximize the reuse of well established protocols and minimize the efforts to deploy our solution.

Eventually, our security analysis successfully yielded all the results we were hoping for, thus confirming the correctness of our proposed extension of the Reissuance Protocol. Now, from all the proofs of correctness generated by the Tamarin prover, we have obtained the formal guarantee that we can securely work out the problem of bootstrapping TLS trust from AS Certificates with a readily deployable solution which only requires minimal changes to the existing SCION infrastructure. This solution not only allows us to establish an encrypted QUIC channel between a BS and an HPS in the context of Hidden Path communication, but it can also inherently scale to provide mutual authentication for any two entities in SCION.

We expect possible future work to build upon the Tamarin models that we have made available with this thesis, so as to assess the security of further extensions of the Reissuance Protocol. This protocol is a cardinal mechanism in SCION's CP-PKI and we speculate it may still be worth of analysis, for instance, in the broader context of certificate revocations and of *PISKES* [13], i.e., a symmetric-key derivation system which was proposed as an addition to the SCION architecture.

Another possibility could be to stretch Tamarin's symbolic modeling capabilities to perform a more sophisticated security analysis of the Reissuance Protocol, e.g., by verifying more advanced security properties or by devising more elaborate adversary models. A great example of how to conduct this kind of advanced security analysis is provided by Jackson at al. [10]. This paper introduces a more refined symbolic representation of signatures in Tamarin, which allows us to capture the subtle and unexpected behaviors of some cryptographic functions affecting several real-world signature schemes. The security properties that we have successfully verified using Tamarin's built-in signing functions could therefore be explored again in light of the more expressive signing functions suggested in the paper. Here, the goal would be to also account for the actual signature schemes employed in the current implementation of the Reissuance Protocol in the attempt to discover attacks which exploit specific weaknesses of their signing algorithms.

Lastly, Tamarin's automated reasoning could be enhanced to prevent the inefficiency in its symbolic backwards search that we have highlighted in Section 6.3, on pages 56 and 57. There, we have explained how, in specific circumstances, the Tamarin prover fails to foresee that certain fresh values must inevitably come from different `Fr` facts, and hence cannot be the same value. For our models, we were able to avoid the unnecessary trace exploration which ensues from this shortcoming by using the inequality restriction introduced in Listing 6.7 and thus by adding suitable action facts with fact symbol `Inequal` to all rules where we found the need to explicitly distinguish between fresh values that we can immediately tell can never be the same. Nevertheless, it would be beneficial to have this issue directly addressed in the proof search algorithm, so as to systematically improve the overall efficiency of future security analyses in Tamarin.

Bibliography

[1] Anapaya Systems AG website. https://www.anapaya.net/about. Accessed: 2020-03-04.

[2] SCION Architecture website. https://www.scion-architecture.net/. Accessed: 2020-02-19.

[3] Tamarin Prover website. https://tamarin-prover.github.io/. Accessed: 2020-02-19.

[4] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Symbolically Analyzing Security Protocols Using Tamarin. *ACM SIGLOG News*, 4(4):19–30, November 2017.

[5] David Basin, Jannik Dreier, and Ralf Sasse. Automated Symbolic Proofs of Observational Equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS 15, pages 1144–1155, New York, NY, USA, 2015. Association for Computing Machinery.

[6] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.*, pages 82–96, 2001.

[7] Cas J F Cremers. The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 414–418, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[8] D. Dolev and A.C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 30(2):198–208, 1983.

[9] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-27, IETF Secretariat, February 2020. http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-27.txt.

[10] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures. Cryptology ePrint Archive, Report 2019/779, 2019. https://eprint.iacr.org/2019/779.

[11] Gavin Lowe. A Hierarchy of Authentication Specifications. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, CSFW 97, page 31, USA, 1997. IEEE Computer Society.

[12] Adrian Perrig, Pawel Szalachowski, Raphael M. Reischuk, and Laurent Chuat. *SCION: A Secure Internet Architecture.* Springer International Publishing AG, 2017.

[13] Benjamin Rothenberger, Dominik Roos, Markus Legner, and Adrian Perrig. PISKES: Pragmatic Internet-Scale Key-Establishment System. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS'20)*, 2020.

[14] B. Schmidt, R. Sasse, C. Cremers, and D. Basin. Automated Verification of Group Key Agreement Protocols. In *2014 IEEE Symposium on Security and Privacy*, pages 179–194, 2014.

[15] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF 12, pages 78–94, USA, 2012. IEEE Computer Society.

[16] Scionproto. Bootstrap TLS trust from AS Certificates. Issue 2882. https://github.com/scionproto/scion/issues/2882. Accessed: 2020-03-04.

[17] Scionproto. Hidden Paths Design. https://github.com/scionproto/scion/blob/master/doc/HiddenPaths.md. Accessed: 2020-04-01.

[18] Scionproto. Hidden Paths Design - Control Plane messages specification. https://github.com/scionproto/scion/blob/master/proto/path_mgmt.capnp. Accessed: 2020-03-04.

[19] Martin Thomson and Sean Turner. Using Transport Layer Security (TLS) to Secure QUIC. Internet-Draft draft-ietf-quic-tls-10, IETF

Secretariat, March 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-tls-10.txt.

Abbreviations

AS Autonomous System

BS Beacon Server

CA Certificate Authority
CP-PKI Control Plane PKI
CS Certificate Server

HPG Hidden Path Group
HPGCfg Hidden Path Group Configuration
HPS Hidden Path Server

ISD Isolation Domain

PCB Path-segment Construction Beacon
PKI Public-Key Infrastructure

TRC Trust Root Configuration

Glossary

Autonomous System (AS) A locally connected network under a common administrative control

Beacon Server (BS) A server responsible for the SCION path exploration mechanism

Certificate Authority (CA) An entity trusted to identify entities in a network and to bind them to public keys by issuing digital certificates

Certificate Server (CS) A server that keeps cached copies of Trust Root Configurations (TRCs) and SCION Certificates

Hidden Path Group (HPG) A gouping of ASes sharing hidden path information among them

Hidden Path Group Configuration (HPGCfg) The configuration file defining a Hidden Path Group (HPG)

Hidden Path Server (HPS) A server responsible for caching hidden segments and answering hidden path requests

Isolation Domain (ISD) A hierarchical grouping of networks under a common organizational domain sharing the same TRC

Path-segment Construction Beacon (PCB) Cryptographically protected messages generated by Primary ASes and spread across the network to collect routing path information

Path Server A server responsible for the SCION path registration and lookup mechanisms

Trust Root Configuration (TRC) A certificate that defines the roots of trust (i.e., public keys) for an Isolation Domain (ISD)

Appendix A

TRC Structure

In SCION, a Trust Root Configuration (TRC) is made up of a `payload` field and a `signatures` field. The former points to the payload of the TRC, while the latter points to an array of signatures computed over the TRC payload and paired with their metadata.

The structure of the TRC payload comprises the following fields:

- **`isd`**: Isolation Domain (ISD) identifier (unique and immutable).
- **`version`**: TRC version number (starts at 1).
- **`base_version`**: version number of the base TRC.
- **`description`**: human-readable description of the ISD or of the TRC.
- **`voting_quorum`**: number of Primary ASes with voting privileges needed for signing a TRC update.
- **`format_version`**: version number of the TRC format.
- **`grace_period`**: number of seconds during which the unexpired previous version of the TRC is still considered active.
- **`trust_reset_allowed`**: boolean value specifying if a trust reset for the ISD is allowed.
- **`validity`**: a `validity` object (see Section A.1).
- **`primary_ases`**: object mapping the identifier of each Primary AS in the ISD, expressed as an ISD-AS identifier, to an `attributes` object (see Section A.2) and a `keys` object (see Section A.3).
- **`votes`**: object mapping the ISD-AS identifier of each AS that signed the TRC update to a `signature` object (see Section A.4).
- **`proof_of_possession`**: object mapping ISD-AS identifiers to an array of possible key types (`offline`, `online`, and `issuing`). This object

identifies all keys that appear new or updated in the current version of the TRC. ASes owning the corresponding private keys need to additionally sign the TRC with these keys to show *proof of possession* for them.

A.1 The `validity` object

The structure of a `validity` object within a TRC comprises the following fields:

- **`not_before`**: timestamp before which the containing TRC is not considered valid yet.
- **`not_after`**: timestamp after which the containing TRC is not considered valid anymore.

A.2 The `attributes` object

The `attributes` object represents a set of attributes reflecting privileges that a Primary AS may have. These attributes are: `authoritative`, `core`, `issuing`, and `voting`.

A.3 The `keys` object

The `keys` object maps up to three key types (`offline`, `online`, and `issuing`) to an object with the following three fields:

- **`key_version`**: key version number (starts at 1).
- **`algorithm`**: identifier of the algorithm this keys can be used with.
- **`key`**: the corresponding public key.

A.4 The `signature` object

The structure of a `signature` object within a TRC comprises the following fields:

- **`key_type`**: type of the key used to sign the TRC update (either `offline` or `online`).
- **`key_version`**: version number of the key used for producing the signature.

Appendix B

SCION Certificates Structure

The structure of an Issuer Certificate and of an AS Certificate is identical, except for the contents of the `issuer` object (see Section B.3). This is because an AS Certificate is signed with an *Issuer Certificate Key*, and hence authenticated by an Issuer Certificate, while an Issuer Certificate is authenticated directly by a Trust Root Configuration (TRC), since it is self-signed with an *Issuing Key*.

A SCION Certificate is made up of a `payload` field and a `signature` field. The former points to the payload of the SCION Certificate, while the latter points to the signature computed over the payload, together with its metadata.

The structure of a SCION Certificate payload comprises the following fields:

- **`subject`**: ISD-AS identifier (unique).
- **`version`**: SCION Certificate version number (starts at 1).
- **`format_version`**: version number of the SCION Certificate format.
- **`description`**: human-readable description of the Autonomous System (AS) or of the SCION Certificate.
- **`certificate_type`**: either `issuer` for Issuer Certificates or `as` for AS Certificates.
- **`validity`**: a `validity` object (see Section B.1).
- **`keys`**: a `keys` object (see Section B.2).
- **`issuer`**: an `issuer` object (see Section B.3).

B.1 The `validity` object

The structure of a `validity` object within a SCION Certificate comprises the following fields:

- **`not_before`**: timestamp before which the containing SCION Certificate is not valid for verifying signatures yet.
- **`not_after`**: timestamp after which the containing SCION Certificate is not valid for verifying signatures anymore.

B.2 The `keys` object

The `keys` object maps each type of key included in the SCION Certificate (`issuing`, `encryption`, `signing`, or `revocation`) to an object comprising the following fields:

- **`algorithm`**: identifier of the algorithm this keys can be used with.
- **`key`**: the corresponding public key.
- **`key_version`**: key version number (starts at 1).

Key type `issuing` stands for the *Issuer Certificate Key* of an Issuer Certificate. Key type `signing` stands for the *Signing Key* of an AS Certificate. Key type `revocation` stands for the *Revocation Key*, which is optional both in Issuer and in AS Certificates, and can be used in the *SCION Certificate revocation* process. Lastly, key type `encryption` stands for the *Encryption Key*. This key can only be found in AS Certificates and is part of a symmetric-key derivation system called *PISKES* [13] which was proposed as an extension to the SCION architecture.

B.3 The `issuer` object

If field `certificate_type` is set to `as`, then the `issuer` object comprises the following fields:

- **`ia`**: ISD-AS identifier of the Issuing AS signing this AS Certificate with an Issuer Certificate Key.
- **`certificate_version`**: version number of the Issuer Certificate whose associated Issuer Certificate Key was used for signing this AS Certificate.

Otherwise, if field `certificate_type` is set to `issuer`, then the `issuer` object only comprises the following field:

- **`trc_version`**: version number of the TRC that the Issuing AS used when self-signing this Issuer Certificate with its associated Issuing Key.

Appendix C

AS Certificate Reissuance Protocol

The *AS Certificate Reissuance Protocol*, here simply referred to as the *Reissuance Protocol*, is a protocol executed between a *Requesting AS* and an *Issuing AS* located in the same Isolation Domain (ISD). The goal of this protocol is to enable the Issuing AS to respond to a request for AS Certificate renewal from the Requesting AS with a newly issued AS Certificate version.

C.1 Initial Knowledge

Both the Requesting AS and the Issuing AS need to hold some initial knowledge before they can initiate the protocol.

Their shared knowledge comprises:

- The current time from a clock synchronized with all other ASes in the ISD, which we are going to refer to as `NOW`.
- All TRCs issued for the ISD, starting from the current base TRC which bootstrapped the update chain.

The Issuing AS is also expected to know:

- All AS Certificates issued for the Requesting AS.
- All Issuer Certificates referencing the Issuing AS in their `subject` field and the Issuer Certificate Keys associated with them.

On the other hand, the Requesting AS is expected to know:

- A Certificate Chain `<Iss_Cert_version_w, AS_Cert_version_v>`, where:
  - `AS_Cert_version_v` is the latest AS Certificate (version `v`) in store referencing the Requesting AS in its `subject` field.
  - `Iss_Cert_version_w` is the Issuer Certificate (version `w`) referenced in `AS_Cert_version_v.payload.issuer`.

  - `AS_Cert_version_v.payload.validity.not_before`
    `<= NOW <= AS_Cert_version_v.payload.validity.not_after`.

- The Signing Key associated with `AS_Cert_version_v`.

C.2 Full Formal Specification

The Reissuance Protocol consists of three phases, called, in order of execution, *Requesting AS Send*, *Issuing AS Receive and Send*, and *Requesting AS Receive*.

Below, we are going to provide the formal specification for these three phases, which lists into detail all of their fundamental execution steps.

1. Requesting AS Send.

1. Create new `Request` as the copy of `AS_Cert_version_v`.
2. Increment `Request.payload.version` by `1`.
3. (Optional) Update Signing Key:
   a) Update `Request.payload.keys.signing.algorithm` and `Request.payload.keys.signing.key`.

   b) Increment `Request.payload.keys.signing.version` by `1`.
4. Produce signature `Sign_Req_new_key` over `Request.payload` with the (possibly new) Signing Key associated with `Request`.
5. Set `Request.signature` to `Sign_Req_new_key`.
6. Produce signature `Sign_Req_old_key` over `Request` with the Signing Key associated with `AS_Cert_version_v`.
7. Send `<Request, Sign_Req_old_key>` to the Issuing AS.

2. Issuing AS Receive and Send.

1. Receive `<Request, Sign_Req_old_key>` from the Requesting AS.
2. Retrieve Certificate Chain `<Iss_Cert_version_y, AS_Cert_version_x>`, where:
   a) `AS_Cert_version_x` is the latest AS Certificate (version `x`) issued for the Requesting AS.

   b) `Iss_Cert_version_y` is the Issuer Certificate (version `y`) referenced in `AS_Cert_version_x.payload.issuer`.

   c) `AS_Cert_version_x.payload.validity.not_before <= NOW <= AS_Cert_version_x.payload.validity.not_after`.

3. Check `Request.payload.version`:

   a) If `(Request.payload.version - 1) == x`, **continue execution**.

   b) If `Request.payload.version <= x`, return the Certificate Chain `<Iss_Cert_version_y, AS_Cert_version_x>` to the Requesting AS and **stop execution**.

   c) If `(Request.payload.version - 1) > x`, signal a malformed reissuance request and **stop execution**.

4. Verify `Sign_Req_old_key` with `AS_Cert_version_x.payload.keys.signing.key`.

5. Validate `Request`:

   a) Check `Request.payload.subject == AS_Cert_version_x.payload.subject`.

   b) Check `Request.payload.issuer.ia == own ISD-AS identifier`.

   c) Check `Request.payload.issuer.certificate_version == y`.

   d) Check `(Request.payload.keys.signing.key !== AS_Cert_version_x.payload.keys.signing.key) iff (Request.payload.keys.signing.key_version == AS_Cert_version_x.payload.keys.signing.key_version + 1)`.

6. Verify `Request.signature` with `Request.payload.keys.signing.key`.

7. Issue a new Certificate Chain `<Iss_Cert_version_z, Reply>`, where:

   a) `Iss_Cert_version_z` is the latest Issuer Certificate (version `z`) in store, and:

      i. `z >= y`.
      ii. `Iss_Cert_version_z.payload.validity.not_before <= NOW <= Iss_Cert_version_z.payload.validity.not_after`.

   b) `Reply` is a new object created as the copy of `Request`, but where:

      i. `Reply.payload.validity` is updated to satisfy `Iss_Cert_version_z.payload.validity.not_before <= Reply.payload.validity.not_before <= NOW <= Reply.payload.validity.not_after <= Iss_Cert_version_z.payload.validity.not_after`.
      ii. `Reply.payload.issuer.certificate_version` is set to `Iss_Cert_version_z.payload.version`.

c) `Reply.signature` is set to the signature produced over `Reply.payload` with the Signing Key associated with `Iss_Cert_version_z`.

8. Store `<Iss_Cert_version_z, Reply>` as a newly issued Certificate Chain.
9. Send `<Iss_Cert_version_z, Reply>` to the Requesting AS.

3. Requesting AS Receive.

1. Receive `<Iss_Cert_version_z, Reply>` from the Issuing AS.
2. Check `Request.payload.keys.signing` == `Reply.payload.keys.signing`.
3. Check `Request.payload.issuer.ia` == `Reply.payload.issuer.ia`.
4. Check `Request.payload.issuer.certificate_version <= Reply.payload.issuer.certificate_version`.
5. Retrieve `TRC_version_t`, i.e., the latest TRC (version t) in store.
6. Check that `TRC_version_t` is active, i.e., `TRC_version_t.validity.not_before <= NOW <= TRC_version_t.validity.not_after`.
7. Validate the received Certificate Chain `<Iss_Cert_version_z, Reply>`:

   a) `Iss_Cert_version_z.subject` must be listed as an Issuing AS in `TRC_version_t`.

   b) Retrieve `TRC_version_q`, i.e., the TRC (version q) referenced in `Iss_Cert_version_z.payload.issuer.trc_version`.

   c) Check `TRC_version_q.version <= TRC_version_t.version`.

   d) Check `TRC_version_q.validity.not_before <=`
   `Iss_Cert_version_z.payload.validity.not_before <=`
   `Reply.payload.validity.not_before <= NOW <=`
   `Reply.payload.validity.not_after <=`
   `Iss_Cert_version_z.payload.validity.not_after <=`
   `TRC_version_q.validity.not_after <= TRC_version_t.validity.not_after`.

   e) Check that `TRC_version_q` and `TRC_version_t` have listed the same `issuing` public key for the Issuing AS.

   f) Validate `Reply`:

   - Check `Reply.subject` == own ISD-AS identifier.
   - Verify `Reply.signature` with `Iss_Cert_version_z.payload.keys.signing.key`.

   g) Validate `Iss_Cert_version_z`:

      - Check `Iss_Cert_version_z.subject` == ISD-AS Identifier of the Issuing AS.
      - Verify `Iss_Cert_version_z.signature` with the `issuing` public key listed in `TRC_version_t` for the Issuing AS.

8. If any of the checks in the previous step failed:

   a) Retrieve `GraceTRC`, i.e. TRC version t - 1.

   b) Check that `GraceTRC` is active, i.e.,
   `(GraceTRC.validity.not_before <= NOW <= GraceTRC.validity.not_after)`
   `AND`
   `(NOW <= TRC_version_t.validity.not_before + TRC_version_t.grace_period)`.

   c) Repeat checks in Step 7 with `GraceTRC` replacing `TRC_version_t`.

9. Store `<Iss_Cert_version_z, Reply>` as a newly issued Certificate Chain.

Appendix D

Abstracted Reissuance Protocol

If we consider all the modeling abstractions presented in Section 5.3, it should be clear that the specification of the *AS Certificate Reissuance Protocol* that we have exhaustively described in Section C of the Appendix needs some substantial adaptations before it can be implemented in Tamarin.

In the course of this section, we are going to formally present how the Reissuance Protocol was adjusted to fit our modeling choices in Tamarin.

D.1 Initial Knowledge

Both the Requesting AS and the Issuing AS still need to hold some initial knowledge before they can initiate the abstracted protocol, but here, their shared knowledge only comprises the unique TRC issued for the ISD.

The Issuing AS is also expected to know:

- All AS Certificates issued for the Requesting AS.
- All Issuer Certificates referencing the Issuing AS in their `subject` field and the Issuer Certificate Keys associated with them.

On the other hand, the Requesting AS is expected to know:

- A Certificate Chain `<Iss_Cert_version_w, AS_Cert_version_v>`, where:
  - `AS_Cert_version_v` is the latest received AS Certificate (version `v`) referencing the Requesting AS in its `subject` field.
  - `Iss_Cert_version_w` is the Issuer Certificate (version `w`) referenced in `AS_Cert_version_v.payload.issuer`.
  - `AS_Cert_version_v.payload.validity` is a validity indicator which has not yet expired.

- The Signing Key associated with `AS_Cert_version_v`.

D.2 Full Formal Specification

Below, we are going to provide a new formal specification of the three phases making up the Reissuance Protocol. This time, we will list into detail all of their execution steps the way we have modeled them in Tamarin following the abstractions presented in Section 5.3.

1. Requesting AS Send.

1. Create new `Request` as the copy of `AS_Cert_version_v`.
2. Increment `Request.payload.version` by `1`.
3. (Optional) Update Signing Key:
   a) Update `Request.payload.signing.key`.
   b) Increment `Request.payload.signing.version` by `1`.
4. Produce signature `Sign_Req_new_key` over `Request.payload` with the (possibly new) Signing Key associated with `Request`.
5. Set `Request.signature` to `Sign_Req_new_key`.
6. Produce signature `Sign_Req_old_key` over `Request` with the Signing Key associated with `AS_Cert_version_v`.
7. Send `<Request, Sign_Req_old_key>` to the Issuing AS.

2. Issuing AS Receive and Send.

1. Receive `<Request, Sign_Req_old_key>` from the Requesting AS.
2. Retrieve Certificate Chain `<Iss_Cert_version_y, AS_Cert_version_x>`, where:
   a) `AS_Cert_version_x` is the latest AS Certificate (version `x`) issued for the Requesting AS.
   b) `Iss_Cert_version_y` is the Issuer Certificate (version `y`) referenced in `AS_Cert_version_x.payload.issuer`.
   c) `AS_Cert_version_x.payload.validity` is a validity indicator which has not yet expired.
3. Check `Request.payload.version`:

   a) If `(Request.payload.version - 1) == x`, **continue execution**.

   b) If `Request.payload.version <= x`, trigger an out-of-band resynchronization process updating the Requesting AS with the missing Certificate Chain `<Iss_Cert_version_y, AS_Cert_version_x>` and then **stop execution**.

   c) If `(Request.payload.version - 1) > x`, **stop execution**.

4. Verify `Sign_Req_old_key` with `AS_Cert_version_x.payload.signing.key`.

5. Validate `Request`:

   a) Check `Request.payload.subject == AS_Cert_version_x.payload.subject`.

   b) Check `Request.payload.issuer.ia == own AS identifier`.

   c) Check `Request.payload.issuer.certificate_version == y`.

   d) Check `(Request.payload.signing.key !== AS_Cert_version_x.payload.signing.key) iff (Request.payload.signing.key_version == AS_Cert_version_x.payload.signing.key_version + 1)`.

6. Verify `Request.signature` with `Request.payload.signing.key`.

7. Issue a new Certificate Chain `<Iss_Cert_version_z, Reply>`, where:

   a) `Iss_Cert_version_z` is its latest Issuer Certificate (version z), and:

      i. `z >= y`.
      ii. `Iss_Cert_version_z.payload.validity` is a validity indicator which has not yet expired.

   b) `Reply` is a new object created as the copy of `Request`, but where:

      i. `Reply.payload.validity` is updated to a new validity indicator.
      ii. `Reply.payload.issuer.certificate_version` is set to `Iss_Cert_version_z.payload.version`.

   c) `Reply.signature` is set to the signature produced over `Reply.payload` with the Signing Key associated with `Iss_Cert_version_z`.

8. Register `Iss_Cert_version_z` as a newly issued Issuer Certificate.

9. Send `<Iss_Cert_version_z, Reply>` to the Requesting AS.

3. Requesting AS Receive.

1. Receive `<Iss_Cert_version_z, Reply>` from the Issuing AS.
2. Check `Request.payload.signing` == `Reply.payload.signing`.
3. Check `Request.payload.issuer.ia` == `Reply.payload.issuer.ia`.
4. Check `Request.payload.issuer.certificate_version` <= `Reply.payload.issuer.certificate_version`.
5. Validate the received Certificate Chain `<Iss_Cert_version_z, Reply>`:
   a) Check that `Reply.payload.validity` is a validity indicator which has not yet expired.

   b) Validate `Reply`:
      - Check `Reply.subject` == own AS identifier.
      - Verify `Reply.signature` with `Iss_Cert_version_z.payload.signing.key`.

   c) Validate `Iss_Cert_version_z`:
      - Check `Iss_Cert_version_z.subject` == AS Identifier of the Issuing AS.
      - Verify `Iss_Cert_version_z.signature` with the `issuing` public key listed in the unique TRC for the Issuing AS.
6. Register `<Iss_Cert_version_z, Reply>` as a newly received Certificate Chain.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

[redacted]

[redacted] *papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*