Formal Verification of the 5G Inter-Operator Signaling Protocol

Master's Thesis
Hochschule Wismar

Author:	Hans Christian Rudolph Seta 3-5-15 158-0095 Setagaya Tokyo, Japan
Matriculation no.:	244404
First examiner:	Prof. Dr.-Ing. habil. Andreas Ahrens Hochschule Wismar
Second examiner:	Prof. Dr.-Ing. Antje Raab-Düsternhöft Hochschule Wismar
Advisor:	Prof. Dr. Alf Zugenmaier Hochschule München
Date of submission:	August 26, 2020

Abstract

This thesis analyzes the formal security properties of the PRotocol for N32 INterconnect Security (PRINS), specified by the 3rd Generation Partnership Project (3GPP) for the purposes of protecting signaling traffic between 5G mobile networks. An emerging technology that is believed to enable a plethora of novel and potentially critical applications, it is mandatory that 5G communication be effectively secured by design. Since inter-operator signaling exhibits specific functional requirements that existing security protocols fail to address, 3GPP defines this purpose-built protocol that has yet to undergo thorough analysis by the broader research community at the time of this writing.

Formal methods have successfully been used to validate and improve several security protocols in the past. The nature of this approach to verifying a system's correctness makes it particularly suited for detecting logical flaws in the underlying design. Semi-automated tools for formal verification further aid the discovery of issues that would be non-trivial to identify by manual analysis. As part of this thesis, the PRINS specification is assessed in detail in order to understand all related message flows and derive the intended security properties. Subsequently, they are transposed into a formalized representation and verified against a model of the protocol for the popular model checker ProVerif.

Although no concrete attacks substantiate from the formal verification, it is shown that the 3GPP specification contains several inconsistencies and, most notably, a lack of clarity about what the protocol is supposed to achieve. This ambiguity may potentially lead to unreliable and therefore insecure implementations. Based on these findings, a number of improvements are suggested that can help make the specification more explicit and easier to understand.

Contents

List of Figures

List of Tables

Glossary

2G is an acronym for the second generation of cellular mobile networks. Global System for Mobile Communications (GSM), a 2G-compliant standard, is commonly used synonymously due to its widespread implementation. Developed by the European Telecommunications Standards Institute in 1991, GSM is still in use in many countries around the world at the time of this writing.

3G is an acronym for the third generation of cellular mobile networks. Universal Mobile Telecommunications System (UMTS), a 3G-compliant standard, is commonly used to describe 3GPP Releases 99 through 7 (i.e. 99, 4, 5, 6, 7). The first mobile standard to be developed by 3GPP, is reuses the 2G/GSM Core Network paired with a redesigned Radio Access Network.

4G is an acronym for the fourth generation of cellular mobile networks. Long Term Evolution (LTE), a 4G-compliant standard, is commonly used to describe 3GPP Releases 8–14. It is the first standard for purely packet-switched mobile network.

5G is the official term used by 3GPP to describe its specification Releases 15 and beyond. An evolutionary change on top of 4G, it features a greater functional decoupling within the Core Network and an updated protocol stack for signaling.

Control Plane also known as Signaling Plane, describes protocol messages used to control the mobile network itself, including subscriber authentication, message routing, and session management.

Core Network describes the part of a mobile network that hosts central functionalities, such as subscriber authentication, and mobility management. It is complementary to the Radio Access Network.

Linear-time Temporal Logic describes a system of rules and modalities referring to time that allows one to „describe and reason about how the truth values of assertions vary with time” (Emerson 1990, p. 1). Such system is linear if every point in time has a defined successor.

Radio Access Network describes the part of a mobile network that transmits/receives radio signals to/from mobile handsets. It is complementary to the Core Network. In 5G, the RAN is comprised of multiple base stations called gNodeB (gNB).

User Plane describes protocol messages that are directly created or consumed by the user/user equipment and are only transported by the mobile network.

Abbreviations

3GPP 3rd Generation Partnership Project

AAD Additional Authenticated Data

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

AKA Authentication and Key Agreement

API Application Programming Interface

ECDSA Elliptic Curve Digital Signature Algorithm

ECK Extended Canetti-Krawczyk

FQDN Fully Quantified Domain Name

GCM Galois/Counter Mode

GSM Global System for Mobile Communications

HTTP Hypertext Transfer Protocol

IP Internet Protocol

IPX IP eXchange

IV Initialization Vector

JOSE JSON Object Signing and Encryption

JSON Javascript Object Notation

JWE JSON Web Encryption

JWS JSON Web Signature

LTE Long Term Evolution

MAC Message Authentication Code

MNO Mobile Network Operator

MRS	Multiset Rewriting System
NF	Network Function
PLMN	Public Land Mobile Network
PRINS	PRotocol for N32 INterconnect Security
REST	REpresentational State Transfer
SCTP	Stream Control Transmission Protocol
SEPP	Security Edge Protection Proxy
SS7	Signaling System 7
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TS	Technical Specification
UMTS	Universal Mobile Telecommunications System

Chapter 1

Introduction

This chapter establishes the problem statement and research interest behind the formal verification of the 5G inter-operator signaling protocol. Section 1.1 describes the motivation to explore this particular problem. Section 1.2 outlines the research questions in scope and general working assumptions. Section 1.3 provides an overview of related work on formal verification of security protocols and mobile network protocols in general. Section 1.4 describes the approach to answer the research questions and the overall structure of this thesis.

1.1 Context and Problem Statement

The fifth generation of mobile networks, 5G, is expected to enable ubiquitous broadband access and a range of novel use cases, such as the massive Internet of Things, lifeline communication, and e-health services (see NGMN Alliance 2015, p. 13). Naturally, as our society grows to be more connected and the applications of this technology become increasingly critical, ensuring its safety and security is vital not only to prevent harm to digital assets but to human lifes as well. Doing so requires comprehensive and continuous efforts that include –among others– secure technical specifications, reliable implementations, as well as secure operations and management. This thesis solely focussed on one part of the 5G specification produced by the 3rd Generation Partnership Project (3GPP), an international consortium of Mobile Network Operators (MNOs), network equipment vendors, regulators, and other stakeholders.

A fundamental requirement for global connectivity in the 3GPP ecosystem is communication between different Public Land Mobile Networks (PLMNs). Whether a subscriber visits a foreign network, commonly known as *roaming*, or locally consumes services that involve parties in different mobile networks (e.g. international calls), mobile network operators need to exchange data with their peers. This communication is routed through a global network called IP eXchange (IPX), separate from the Internet. Originally a private communication channel of the mobile operator community, the privatization of telecommunication providers and competition laws gradually mandated interconnect networks to be opened up to other businesses as well. Today it is known for being a security risk that operators are inevitably exposed to due to a number of shortcomings related to this shift. Firstly, it lacks proper access control to restrict its members to trustworthy parties only, as mobile network operators regularly resell their connectivity as part of partnerships or even end-user offerings. Secondly, the fact that some of the protocols used on the IPX network do not offer basic security features such as authentication or confidentiality natively makes this interface particularly prone to malicious traffic. Thirdly, the network infrastructure itself is operated by third-party companies which have full access to signaling traffic that passes through their peering points. While this enables valuable services to be provided by these companies, there is no definitive way for the communication endpoints to control the extent to which different parties access their traffic. Although the associated risks can be minimized by appropriate security controls, the inherent weaknesses of inter-operator signaling renders it one of the major security and fraud risks for mobile network operators.

3GPP attempts to improve the security of inter-operator signaling in its technical specifications of the 5G system. As part of an overall updated protocol stack, Release 15 and beyond include a new security protocol for the so-called N32 interface (see 3GPP 2019c, pp. 130-146). The PRotocol for N32 INterconnect Security (PRINS) is purposefully designed by 3GPP's Security Working Group to fit the specific requirements of inter-operator signaling, described in detail in section 2.1. Although parts of this protocol build on established technologies such as Transport Layer Security (TLS) and JSON Object Signing and Encryption (JOSE), secure protocol design remains a non-trivial task in which defects are difficult to identify by manual analysis. A prominent negative example is the protocol proposed by Needham and Schroeder in 1978 (Needham and Schroeder 1978). Initially considered secure, it was proven to be flawed 17 years after its inception (Lowe 1996).

One instrument that can facilitate the design of secure systems and algorithms is formal verification – establishing correctness proofs with respect to its requirements by means of formal methods of mathematics. Since this process is based on an abstract model of the system under test it is also known as *model checking*. As Alur explains, „a designer first constructs a model, with mathematically precise semantics [...] and performs extensive analysis with respect to correctness requirements" (Alur 2011, p. 1). In combination with computer-aided verification tools, some of which are reviewed in section 2.2, this approach can help to discover logical flaws that would remain undetected using other static testing methods – the above-mentioned Needham-Schroeder protocol only being one example. It should be noted that formal verification can only ever produce proofs about the model it is applied to. Proving facts about a model that does not accurately reflect the actual system yields limited or even false insights.

In light of the 5G system introducing a new inter-operator signaling protocol, the question arises whether PRINS succeeds to cater to the particular requirements of the N32 interface while simultaneously delivering on its intended security properties.

1.2 Aims and Objectives

This thesis aims to investigate security properties of the 5G inter-operator signaling protocol PRINS with the methods of formal verification. Given the compexity of secure protocol design and the fact that 3GPP more often than not references existing protocols rather than creating its own ones, it is considered beneficial to analyze PRINS in detail. Since model checking allows verification based on a system's specification, or rather a formal model thereof, this work is independent of any particular implementation. Consequently, proving or disproving properties of the model is supposed to result in recommendations to improve the underlying specification. In summary, the objective is to answer the following three research questions:

(1) Is it possible to comprehensively describe the 5G inter-operator signaling protocol with existing modeling techniques?

(2) Does modeling PRINS in a formal manner reveal any inconsistencies that should be addressed by the 3GPP specification?

(3) Based on the model created as part of this work, can current formal verification tools prove or disprove particular security requirements of the PRINS protocol?

As the protocol to be verified is comprised of several different technologies, some of them already thoroughly tested, certain aspects are considered out of scope. This includes TLS version 1.2 and 1.3, which have been subject to previous formal analysis and are assumed to be perfectly secure in the following. The reader is referred to Horvat 2015, Cremers et al. 2017, and Merwe 2018 who cover this protocol in great detail. The same assumption applies to JSON Web Encryption (JWE) and JSON Web Signature (JWS), which are part of the JOSE suite. Their profiles as specified in 3GPP Technical Specification (TS) 33.210 (3GPP 2019a, pp. 19-20), using the Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) and Elliptic Curve Digital Signature Algorithm (ECDSA) respectively, are considered perfect. Analyses of the security properties of these cryptographic algorithms are provided by McGrew and Viega 2004 as well as Fersch, Kiltz, and Poettering 2016.

Furthermore, this thesis is intentionally restricted to the application of existing software tools for formal verification, rather than extending them or even developing a new one. It can be argued that if model checking is to become a common step in the design and standardization of new security protocols, related software tools need to exhibit general applicability so as to be utilized by the protocol designers without having to create special-purpose ones.

1.3 Related Work

The idea of proving correctness of concurrent systems, such as security protocols, based on a formal representation is not a new one. Basin, Cremers, and Meadows 2018 provide an extensive historical account of this kind of model checking, streching back to the 1980s. This section summarizes foundational theories and key contributions to this specific field of research that remains active to this day. Specifically, previous work directly related to the application of formal verification to mobile network security and its effect on the development of these protocols is discussed. As part of this, examples of model checking software commonly used in recent years are highlighted, which serves as a basis for choosing a particular tool in section 2.2. Lastly, research on inter-operator signaling protocols of previous mobile generations and their security vulnerabilities is cited in order to emphasize this particular subject's relevancy.

The term *model checking* as it is used today goes back to Clarke and Emerson, who are the first to propose the computer-aided verification of finite-state systems expressed in Linear-time Temporal Logic. In contrast to previous efforts, they model the system flow graph as a finite structure and describe „an efficient algorithm [...] to decide whether a given finite structure is a model of a particular formula" (Clarke and Emerson 1981, p. 2). Even though this particular work is used to reason about a concurrent program on a single machine, the authors already hint to the possible extension of their approach to „network communication protocols" (Clarke and Emerson 1981, p. 2). The work of Dolev and Yao (Dolev and Yao 1983) consitutes another landmark paper in terms of formally verifying security protocols. Their contribution is to model a protocol „as a machine consisting of an arbitrary number of honest agents executing the protocol, in which all messages sent are intercepted by the adversary [...], all messages received are sent by the adversary" (Basin, Cremers, and Meadows 2018, p. 6). A foundational concept for many model checking tools to this day, it is shown in section 2.1 how this assumption is an excellent fit for the use case of inter-operator signaling.

The Dolev-Yao model further represents all messages in abstract mathematical terms and assumes the underlying cryptography to be perfect – an approach also known as formal verification in the symbolic model or *symbolic model checking.* This stands in contrast to *computational model checking* in which „messages are bitstrings, and the adversary is a probabilistic polynomial-time Turing machine" (Blanchet 2012, p. 1). Although successfully used for verifying properties of several security protocols (see Blanchet, Jaggard, et al. 2008, Cadé and Blanchet 2013, Lipp, Blanchet, and Bhargavan 2019), one of the basic assumptions of the thesis outlined in the previous section is perfect cryptography. Therefore, this approach is only mentioned for the sake of completeness and shall not be pursued further.

In recent years, 3GPP networks and protocols clearly capture the interest of the model checking research community. This does not just apply to the latest 5G specifications, but to earlier generations as well, such as 3G/UMTS (see Alt et al. 2016) and 4G/LTE (see Lee et al. 2014). However, these papers mostly focus on the Authentication and Key Agreement (AKA) protocol used for mutual authentication between the subscriber's handset and the mobile network. Similar efforts are documented for the slightly modified 5G-AKA protocol. Dehnel-Wild and Cremers analyze a draft version of the 5G security specification and discover a potentially critical race-condition (see Dehnel-Wild and Cremers 2018). By replaying an overheard, encrypted subscription identifier following the authentication of a genuine user with the network, an attacker may be able to induce a session mis-binding in the communication between two 5G core network elements, leading to the leakage of the genuine subscriber's Authentication Vector. Although discussed by 3GPP's Security Working Group, this publication did not directly result in any of the improvements suggested by the authors. Basin, Dreier, Hirschi, et al. provide a comprehensive analysis that highlights a number flaws uncovered using formal methods and suggesting possible improvements (see Basin, Dreier, Hirschi, et al. 2018). Specifically, they point out a lack of explicit specification of desired security properties, confidentiality requirements that are too weak, redundancies in the messages exchanged between network and subscriber, the use of a sequence number for replay protection, and a flawed key confirmation procedure. Borgaonkar et al. demonstrate a proof of concept and formal verification for an attack that utilizes authentication challenges obtained on a victim's behalf to infer at least 10 bits of the sequence number used in 5G AKA (see Borgaonkar et al. 2019). Of these three papers on the 5G system, none of them directly resulted in the improvements the researchers suggested, although having been discussed in 3GPP's Security Working Group (see Vodafone 2018).

The software used to aid the formal proofs on the 5G security protocols above is the TAMARIN prover (see Meier et al. 2013). The logical successor to another popular model checking tool, SCYTHER (see Cremers 2008, Cremers 2020), it is used in a number of research papers on formal verification and enjoys active development. The tool PROVERIF (see Blanchet et al. 2016, Blanchet 2020) takes a slightly different approch to symbolic model checking by describing protocols in a formal process algebra rather than multiset rewriting systems employed by TAMARIN. An in-depth comparison of these two model checking tools is provided in section 2.2.

Since the first 5G specification, i.e. 3GPP Release 15, has officially been released just over two years ago, security research has yet to cover all of its aspects. At the time of this writing, the author is not aware of any work that specifically focuses on the PRINS protocol. However, from efforts on inter-operator signaling in previous mobile generations and the security risks associated with it, one can deduct that it is just a matter of time until broader research interest catches up with the 5G equivalent. Security issues of Signaling System 7 (SS7) were made public in 2008 and have since been well documented (see Engel 2008, Engel 2014, De Oliveira et al. 2014, Puzankov and Kurbatov 2014). Furthermore, Diameter, the inter-operator signaling protocol used in between 4G/LTE network, is proven to share many of these weaknesses with its predecessor (see Rao, Holtmanns, et al. 2015, Rao, Kotte, and Holtmanns 2016, Rao, Oliver, et al. 2016, Holtmanns, Rao, and Oliver 2016, Holtmanns and Oliver 2017). Hence, the European Network and Information Security Agency recognizes a „medium to high level of risk" (ENISA 2018, p. 23) in present-day mobile networks and points to additional troubles in 5G, such as shorter time to real-world exploitation due to the broad use of common web protocols.

1.4 Outline and Methodology

Chapter 2 outlines the theoretical preliminaries implied throughout the remainder of this thesis. Initially, section 2.1 examines the relevant 3GPP specifications in detail to establish a firm understanding of all protocol components, involved participants, messages flows, and desired security properties of PRINS. Particular emphasis is put on the trust model assumed by the 3GPP specification. This first step serves as the basis for transposing the specification into a formal model later on. Secondly, section 2.2 goes on to describe key principles behind symbolic model checking for the purposes of verifying security protocols.

As part of this, a brief introduction to first-order logic is provided, which serves as the fundamental theory to system verification using formal methods in general. It is further described what kind of security properties can be verified with this apporach and what known limitations exist. Subsequently, two popular tools for automated model checking of security protocols, TAMARIN and PROVERIF, are analysed in more detail. In order to determine the software capable of addressing the requirements of the protocol under test in an optimal way, a comparison is performed in terms of supported security properties and cryptographic primitives. Aside from the pure security requirements to be verified and a particular tool's cryptographic feature set, additional aspects such as the channel characteristics and supporting modeling constructs are taken into account as well. The decision to use a particular software tool also determines the model that needs to be created in the following step. While there are attempts to bridge the gap between the aforementioned tools by translating one type of formal model into the other (see Kremer and Künnemann 2016), this thesis is intentionally focused on TAMARIN and PROVERIF in there pure form, without relying on third-party extensions.

Chapter 3 contains a comprehensive description of transforming the existing PRINS specification into a abstract model as well as abstractions and assumptions made during this process. Section 3.1 provides a detailed account of modeling all protocol participants in a way that is suitable to be verified by the tool chosen in the previous step. During this process, the objective is to answer research question (1) conclusively. Since 3GPP documents are not written in a strictly formal manner but rather in prose, the protocol specification may prove not to be perfectly complete – a point of critique Basin, Cremers, and Meadows highlight with respect to the 5G AKA protocol (see Basin, Cremers, and Meadows 2018). In these cases, explicit abstractions and assumptions about the intended protocol behavior or requirement shall be made to allow for the completion of the computer-aided verification. Any such findings contribute to answering research question (2) and are documented in section 3.2. As a result, suggestions for the improvement of the 3GPP specifications are captured in section 4.3.

Chapter 4 discusses the process of computer-aided verification, the obtained results and their security implications. Section 4.1 analyses the issues encountered during execution of the model checking tool as well as the information obtained as a result. Two kinds of findings are expected at this stage: Firstly, potential problems while running the software aid the validation of the model created previously. For example, if the tool does not terminate this may be an indication of the protocol being defined too loosely and a refinement of the model may be required. Secondly, if an actual flaw in the model is discovered it should be possible to explicitly point out the root cause, since tools for symbolic model checking by design are able to provide a counter example to a property in the form of a particular trace that violates it. Section 4.3 puts these findings into context and outlines the practical security implications on the PRINS protocol, if any. Where required, recommendations for improving the protocol's specification are provided.

Lastly, chapter 5 draws a final conclusion of the formal verification of PRINS, summarizes the contributions of this thesis, and provides suggestions for potential future work based on the discovered findings.

Chapter 2

Theoretical Preliminaries

This chapter establishes the theoretical foundations of the 5G inter-operator protocol and formal verification. Section 2.1 describes design rationale and functional requirements of PRINS. Subsequently, the individual protocol components and message flows are outlined, specifically focussing on security requirements to be verified. Section 2.2 serves as an introduction to symbolic model checking of security protocols and compares two tools that may be used for this purpose. Key concepts are explained to an extent that allows the reader to understand the model of the PRINS protocol and its verification later on. Section 2.3 draws a conclusion about which tool to use for further analysis.

2.1 5G Signaling and the PRINS Protocol

2.1.1 Global Interconnect Networks

Ever since the introduction of the first international mobile communication standard 2G/GSM, MNOs around the world exchange signaling and user data via the global IPX network. Designed to be a private channel for communication amongst MNOs only, the way in which it is used today, and the plethora of different parties connected to it –many of them not network operators– renders it a network that is not any more trustworthy than the public Internet. Coupled with weaknesses in the established network protocols as previously summarized in section 1.3, the interconnect interface is cause for major security and fraud concerns for any mobile service provider.

One of the reasons why this well-known problem still persists several years after the publication of possible exploits is the role that operators of IPX peering platforms play in inter-PLMN signaling, which goes well beyond that of pure connectivity providers. Many MNOs depend on them for supplementary services that require accessing or even modifying the contents of signaling traffic. These services include, for example, normalization of the protocol syntax, replacement of attributes to alleviate technical incompatibilities, and business intelligence offerings. Note, that for the communication partners on either side it is intransparent which routing path messages take on the interconnect network. A single PLMN can simultaneously be connected to multiple IPX providers, which in turn may or may not connect directly to each other. 3GPP assumes a model in which there is at least one or more IPX provider in any given routing path (see 3GPP 2019c, p. 130):

– Both PLMNs are connected to the same IPX provider that is allowed to access message contents.

– Both PLMNs are connected to two distinct IPX providers that are each allowed to access message contents and that are connected directly to each other.

– Both PLMNs are connected to two distinct IPX providers that are each allowed to access message contents and that are connected via additional IPX providers not expected to access message contents.

As PLMN operators have a business relationship with their directly connected IPX providers, these entities are considered trusted to the extend that they may supply modifications to incoming and outgoing messages. Similarly, the IPX provider directly connected to the peer operator is trusted by extension, as both MNOs have a business relationship between each other governing the allowed message modifications. Lastly, third-party IPX providers in between the ones directly connected to either of the mobile networks should not modify any message components or access confidential information.

Intuitively, one can dicern an overlap with the premises of the Dolev-Yao model: All signaling messages are intercepted or potentially blocked by an IPX provider, i.e. the potential adversary, and all messages received by any mobile network operator are sent by such an entity. This constellation, in which different external parties are expected to perform certain changes that cannot be validated or traced with existing security protocols is one of the motivating factors for the definition of PRINS. It is supposed to provide the technical controls to enforce the trust model outlined above.

2.1.2 5G Core Network Signaling

3GPP Release 15 features a redesigned protocol stack for communication between 5G Core Network functions, illustrated in figure 2.1. Rather than relying on specialized protocols that are almost exclusively used in previous generations of mobile networks, 5G makes broad use of protocols, formats, and principles popular in the domain of web services. It is important to note that the 5G User Plane is exempt from these changes and is still relying on the same protocols as in 4G/LTE. The 5G Control Plane however, has been updated as follows. The Transmission Control Protocol (TCP) is used on transport layer, rather than the Stream Control Transmission Protocol (SCTP). On top of that, Internet Protocol (IP) packets carry Hypertext Transfer Protocol (HTTP)/2 messages that may optionally be protected by TLS. Inside the HTTP body additional information structured as Javascript Object Notation (JSON) objects can be transferred. Communication between 5G Core Network Functions (NFs) evolves around standardized Application Programming Interfaces (APIs) that are modeled according the principles of

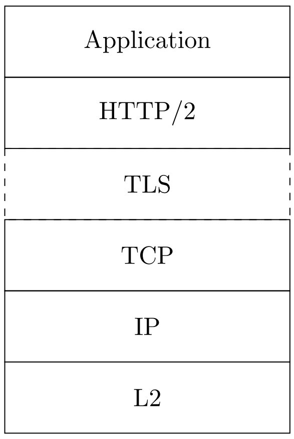


Figure 2.1: 5G control plane stack, according to TS 29.573 (3GPP 2019b, p. 11)

REpresentational State Transfer (REST). This design decision brings with it a number of constrains originally layed out by Fielding and Taylor 2000. The ones adopted by 3GPP in Release 15 are summarized below (see Mayer 2018, p. 109-114):

1) *Resources, Resource Identifiers, and Resource Representation*

Every single data point that might be the target of a signaling message is considered a resource. A resource is an abstract entity that may be referenced by identifiers. Its information can be conveyed in varying representations, e.g. a JSON object.

2) *Client-Server Communication*

Communication follows a strict separation of concerns, allowing both client and server to evolve independently as long as the defined APIs are used. In the 3GPP specification the terms consuming NF and producing NF are equally used.

3) *Statelessness*

Signaling messages are expected to be completely self-contained, allowing the receiving party to intepret them without any additional state information stored across multiple requests/responses. Although this holds true for HTTP itself, both TLS and PRINS being stateful protocols do not adhere to this constraint.

4) *Cache*

Resulting from item 3) is the ability of the client or intermediaries to store certain responses temporarily for future (re-)use. A possible way to optimize intra-PLMN signaling latency, this behaviour is not relevant for inter-operator communication.

5) *Layered Systems*

This constraint demands that the client does not need to be aware of what exactly happens on server side to fulfill a request or even what server instance is answering. In essence, this is about decoupling of individual services which is inherent to the design of the 5G Core Network.

The above-mentioned changes to the protocol stack also extend to the signaling channel between PLMNs, i.e. the N32 interface. But aside from modernizing the protocols, a redesigned Core Network architecture entails further improvements to enhance interconnect security. Whereas earlier mobile generations only specify protection on the transport layer by means of IPsec, which in practice is rarely used due to operator requirements cited in the previous section, 5G aims to offer more flexible protection on application layer. For this purpose, a dedicated Network Function is introduced to ensure security on the N32 interface. The Security Edge Protection Proxy (SEPP) is tasked with applying confidentiality and integrity protection to outbound signaling messages, depending on operator-controlled protection policies. Simultaneously, it is the single point of contact for all inbound signaling traffic, ensuring mutual authentication with all peer SEPPs, enforcing message integrity, rate limits, and further security checks. A complete list of SEPP requirements is given in TS 33.501 (3GPP 2019c, p. 31).

When an NF needs to exchange Control Plane messages with an entity in a different PLMN, it addresses the SEPP in its own network. The SEPP transforms the complete HTTP message into a JSON object before applying security on the individual information elements contained. Protection is provided by means of JWE and symmetric keys shared between peer SEPPs. Afterwards, the transformed message sent on the N32 interface towards the roaming partner's SEPP via one or more IPX providers. Authorized IPX providers may add changes to the transported messages, using the JSON patch method specified in RFC 6902 (Bryan and Nottingham 2013). Each patch is digitally signed using JWS and the IPX provider's private key. Before any message modifications are possible, the related public key needs to be shared with the involved MNOs. By relying on asymmetric cryptography, traceability of any intermediary patches can be ensured. Once a SEPP receives an incoming message on the N32 interface, it validates the integrity of the original contents and any patches added by IPX providers. If this validation is successful, the added JSON patches originate from authorized sources and only modify information elements that are allowed to be accessed, the SEPP transforms the JSON element back into a signaling message, applying changes by the IPX providers in the process. The resulting HTTP message is forwarded to the destination NF. This flow is illustrated in figure 2.2 below.

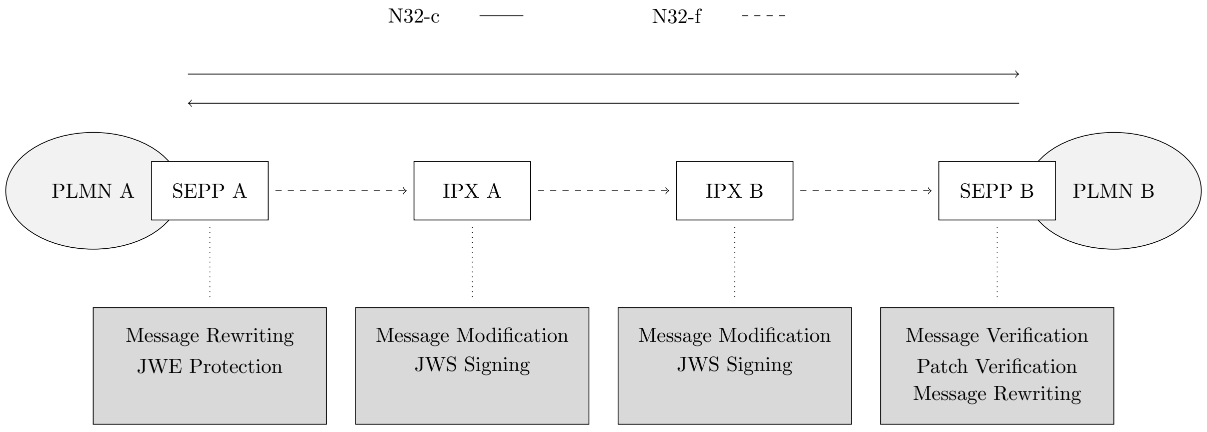


Figure 2.2: 5G interconnect architecture, according to TS 33.501 (3GPP 2019c, p. 131)

2.1.3 PRINS Protocol Structure

Protocol Channels

The 5G inter-operator signaling protocol provides security for messages on the N32 interface between two SEPPs. The protocol is comprised of two logical channels.

N32-c is a TLS-protected connection used for session management. Once an interconnect session is established, each of the communicating SEPPs sets up an N32-c channel to be able to send and receive related messages by its peer. Information exchanged via this channel serve the purpose of session parameter exchange (e.g. protection policies and cryptographic profiles), error handling throughout the communication, as well as session termination. Since none of these use cases require access to the message contents by any intermediaries, messages are transported via this dedicated end-to-end protected channel.

If none of the operators on either end of the communication have a need for the active involvement of IPX providers, 3GPP also specifies the option to utilize the N32-c channel directly for transfering signaling messages without the use of PRINS. This purely TLS-based operation is mentioned only for completeness sake and shall not be analyzed in detail.

N32-f is a PRINS-protected connection carrying HTTP messages that contain the inter-operator signaling data serialized into JSON objects. The level of protection provided for individual information elements in these messages is determined by two types of protection policies, specified by the communicating parties:

– *Data-type encryption policy*, describing what types of information elements require confidentiality protection.

– *Modification policy*, describing what information elements are allowed to be modified by intermediaries IPX providers.

The data-type encryption policy is supported by an *NF API data-type placement mapping*, defining the data-type of a particular information element. The PRINS protocol expects this policy to be the same in both communicating SEPPs in order to rule out inconsistencies in confidentiality requirements: „The data-type encryption policies in the two partner SEPPs shall be equal to enforce a consistent ciphering [...]" (3GPP 2019c, p. 135). In addition to the operator-defined data-type encryption policies, 3GPP defines a set of data-types to be confidentiality protected mandatorily. That is, data of type „Authentication Vectors", „Location data", and „Cryptographic material" (3GPP 2019c, p. 31). None of the custom protection policies compiled by the operators should contradict these default protection requirements. However, the specification does not describe how a potential misconfiguration is to be handled by either of the SEPPs.

The modification policy of each communicating party is only applicable to the IPX provider that the defining MNO has a business relationship with, i.e. any mobile network operator can only actively allow changes by its directly connected interconnection providers. By default, any modifications by intermediaries are prohibited. The policy is roaming partner specific, i.e. one needs to be defined peer PLMN/IPX provider combination.

An N32-f session is identified by an *N32-f context* – supplementary session information stored by each SEPP in order to correlate and validate related messages. This information can be summarized as follows:

– *N32-f context ID*, uniquely identifying a given N32-f session.

– *N32-f peer information*, containing the peer PLMN ID, peer SEPP ID, and its IP address.

– *N32-f context information*, containing details on session validity and an indication whether to use PRINS or TLS.

- *N32-f security context*, including the PRINS session keys, JOSE cipher suites, protection policies, session counters, initialization vectors, and a so-called IPX security information list.

The N32-f context ID is created by both SEPPs during the initial handshake procedure described in the following subsection. It is referenced in each N32-f message in order to inform the receiving SEPP which session to correlate it to.

The N32-f peer information holds the PLMN ID and an additional SEPP ID in order to differentiate multiple instances in the foreign network. A remote SEPP address is required for routing purposes.

The N32-f security context further references the aforementioned protection policies, as their scope is one particular N32-f session. The agreed cryptographic algorithms for JWE protection are stored as cipher suites. Lastly, the N32-f security context holds an IPX security information list – identifiers of all intermediary IPX providers as well as the related public keys or certificates used to validate message modifications submitted by them. If certificates are being used, it is expected that MNOs act as certificate authorities for their IPX providers, issuing certificates for them to sign their message modifications.

Also contained in the N32-f security context is the N32-f key hierarchy, which is comprised of a single master key generated from the initial N32-c TLS session, using the TLS export mechanism specified in RFC 5705 (Rescorla 2010), as well as two sets of session keys and Initialization Vector (IV) derived form the master key and sequence counters. The latter are used for applying JWE protection to N32-f messages – keys for encryption, IVs and counters for constructing the nonce required by AES-GCM. Two sets of each are required since N32-f, similar to N32-c, requires setting up two distinct HTTP connections to enable a SEPP to both send and receive messages. Hence, 3GPP specifies *parallel* session keys and IVs for use by the SEPP that initialted the N32-c session. The other sets of *reverse* session keys and IVs are to be used for a PRINS session with the SEPP that responded to the initial N32-c establishment (see 3GPP 2019c, p. 141-142):

- `parallel_request_key`
- `parallel_response_key`
- `parallel_request_iv_salt`
- `parallel_response_iv_salt`

- `reverse_request_key`
- `reverse_response_key`
- `reverse_request_iv_salt`
- `reverse_response_iv_salt`

N32-f Message Format

Signaling messages sent towards receivers in a different PLMN are reformatted into a JSON structure by the SEPP before being forwarded on the N32 interface. Depending on the protection requirements described in the configured policies, individual parts of the original HTTP message are placed in different locations. 3GPP defines two objects to be created by the sending SEPP. An overview of the complete message structure is provided in figure 2.3.

– *dataToIntegrityProtect* holds all information to be integrity protected. This includes the complete messages with all its HTTP headers, pseudo-headers (e.g. HTTP method), and the HTTP message body with the exception of information elements requiring confidentiality protection, whose values are replaced by null. The SEPP further supplies additional N32-f specific meta data, i.e. N32-f message ID, N32-f context ID, and an ID of the directly connected IPX provider.

– *dataToIntegrityProtectAndCipher* contains information to be both integrity and confidentiality protected. The SEPP creates a JSON patch for each of the concerned elements of the original message and such that there is a patch replacing each of the null values in the *dataToIntegrityProtect* object.

Following the reformatting, both of these JSON objects are protected using JWE. For this purpose, 3GPP only specifies encryption methods that build on Authenticated Encryption with Associated Data (AEAD), i.e. cryptographic schemes ensuring both confidentiality and authenticity of data. The use of JWE with these particular cipher suites allows for the choice to protect integrity and confidentiality of certain objects while others are only integrity protected. Resulting from this operation is the ciphertext itself and a JWE authentication tag, which can be used to verify the integrity of both the ciphertext and the Additional Authenticated Data (AAD).

IPX providers performing intermediate changes are only allowed to do so on the clear text parts of the N32-f message, i.e. the *dataToIntegrityProtect* object. Modifications are not applied directly, but appended to the original message as a separate JSON patch. Before forwarding the message to the next hop, the complete *modifiedDataToIntegrityProtect* object is signed using JWS and the private key belonging to the IPX provider's certificate issues by its client MNO.

– *modifiedDataToIntegrityProtect* contains the operations to be performed on the original message's *dataToIntegrityProtect* object. In addition, the IPX provider supplies a unique identity that is not further defined. Lastly, the JWE authentication tag created by the sending SEPP is included as well, serving the purpose of replay protection.

2.1.4 PRINS Sequences and Security Properties

Following the individual components of the PRINS protocol, the following section provides a step-by-step description of the complete N32-c and N32-f message flows as specified in TS 33.501 (see 3GPP 2019c, pp. 129, 141-143). In preparation of the formal verification in chapter 4, the protocol's intended security properties are specifically highlighted. It should be noted that 3GPP specifications, being written in an informal style, may not explicitly spell out all security requirements to be met by the PRINS protocol. Therefore, a distinction is made between security properties which are explicitly cited in the document (hereafter highlighted in blue) and those that can be derived implicitly from the its contents (hereafter highlighted in gray).

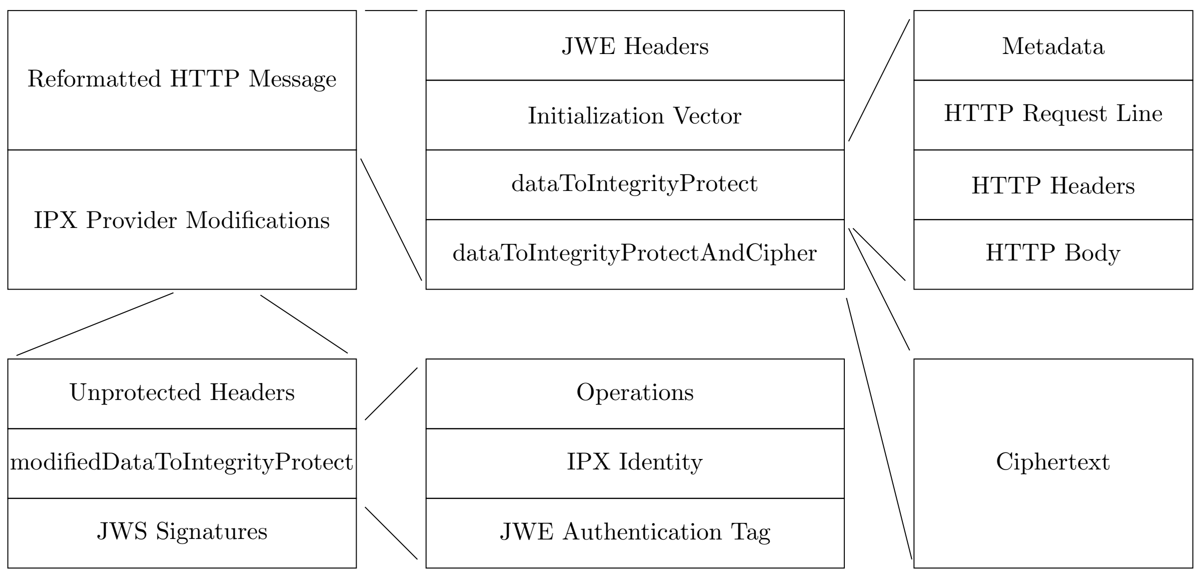


Figure 2.3: N32-f message structure according to TS 29.573 (3GPP 2019b, p. 21)

🔓 Explicitly defined security property	🔓 Implicitly defined security property

N32-c

1. Initially, two peer SEPPs establish an end-to-end protected TLS session on the basis of the previously exchanged raw public keys or certificates for authentication.

> 🔓 Security property 1: N32-c sessions between two SEPPs shall be mutually authenticated.

> 🔓 Security property 2: Information exchanged in an N32-c session shall be confidentiality protected and integrity protected.

2. The initiating SEPP sends an ordered list of supported JOSE cipher suites and its N32-f precontext ID in a `Security_Parameter_Exchange_Request` to its peer.

3. The responding SEPP selects one of the provided cipher suites based on its locally configured priority.

4. The responding SEPP answers the preceding message with a `Security_Parameter_Exchange_Response` containing the highest priority cipher suite supported by both SEPPs and its own N32-f precontext ID.

5. Both SEPPs create the final N32-f context ID by concatenating the initiating SEPP's precontext ID with the the responding SEPP's precontext ID, in that order.

> 🔓 Security property 3: Following above step 5, both SEPPs shall agree on the same N32-f context ID value.

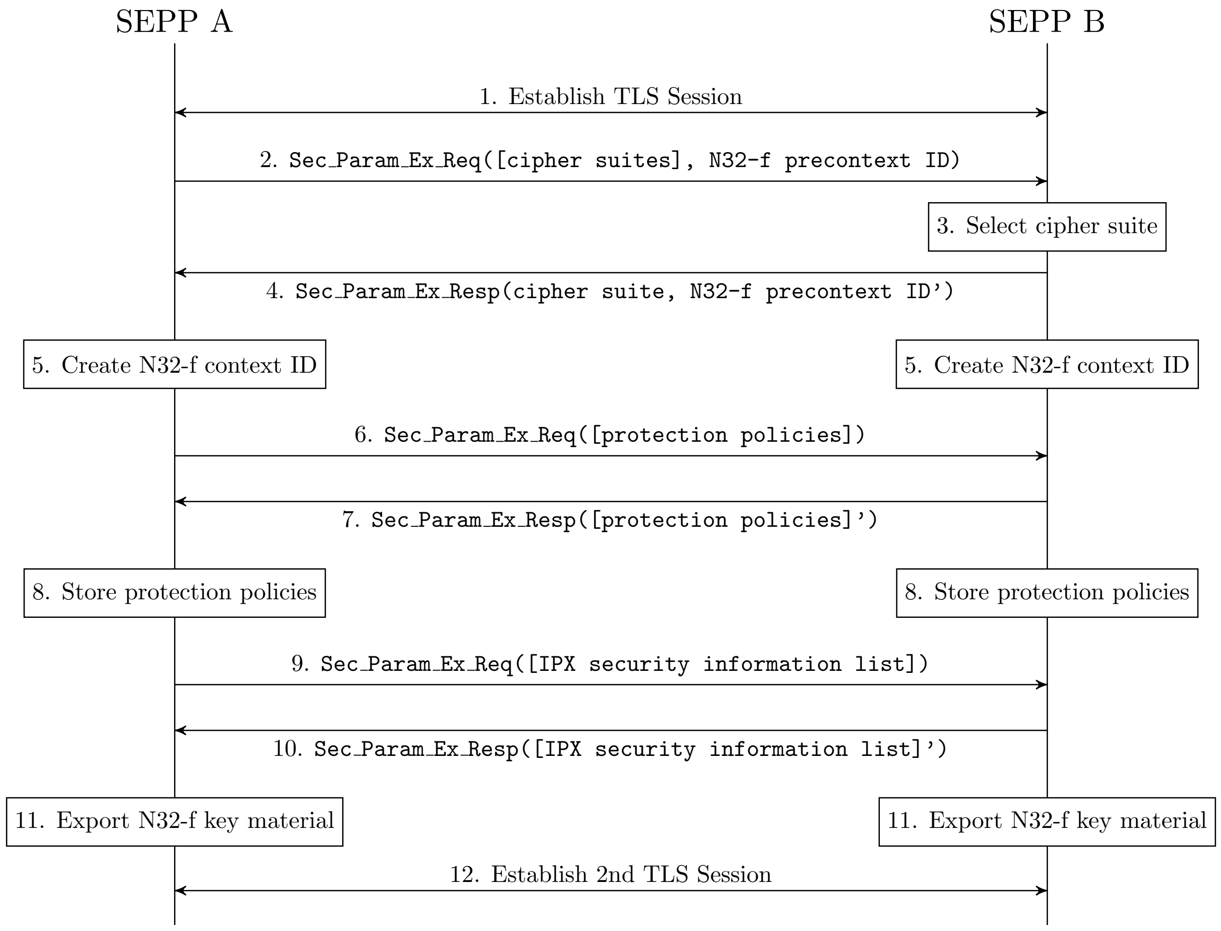

SEPP A
SEPP B
1. Establish TLS Session
2. Sec_Param_Ex_Req([cipher suites], N32-f precontext ID)
3. Select cipher suite
4. Sec_Param_Ex_Resp(cipher suite, N32-f precontext ID')
5. Create N32-f context ID
5. Create N32-f context ID
6. Sec_Param_Ex_Req([protection policies])
7. Sec_Param_Ex_Resp([protection policies]')
8. Store protection policies
8. Store protection policies
9. Sec_Param_Ex_Req([IPX security information list])
10. Sec_Param_Ex_Resp([IPX security information list]')
11. Export N32-f key material
11. Export N32-f key material
12. Establish 2nd TLS Session

Figure 2.4: N32-c Sequence Diagram

6. The initiating SEPP may send its protection policies to its peer. Alternatively, both data-type encryption policies and modicifation policies can also be preconfigured manually.

7. Similarly, the responding SEPP may send its own protection policies as well.

8. If one of the SEPPs exchanged its protection policies in the previous stops, the counterparty shall store these policies in its N32-f security context.

9. The initiating SEPP sends the IPX security information list to its peer. This list contains information about the IPX provider(s) the SEPP's operator has authorized to perform message modifications.

10. Equally, the same information about the IPX provider(s) authorized by the responding SEPP's operator are provided in the opposite direction.

11. Both SEPPs use the TLS export function to derive pseudo-random key material for the N32-f session.

Security property 4: Following above step 11, both SEPPs shall derive the same N32-f master key.

12. The responding SEPP creates a separate TLS-protected N32-c session, enabling it to actively send messages to the initiating SEPP. Both SEPPs leave their N32-c channels open throughout the complete lifetime of a session for the purposes of secure error signaling and session management.

Security property 5: An adversary shall not be able to detect a change in session keys.

N32-f

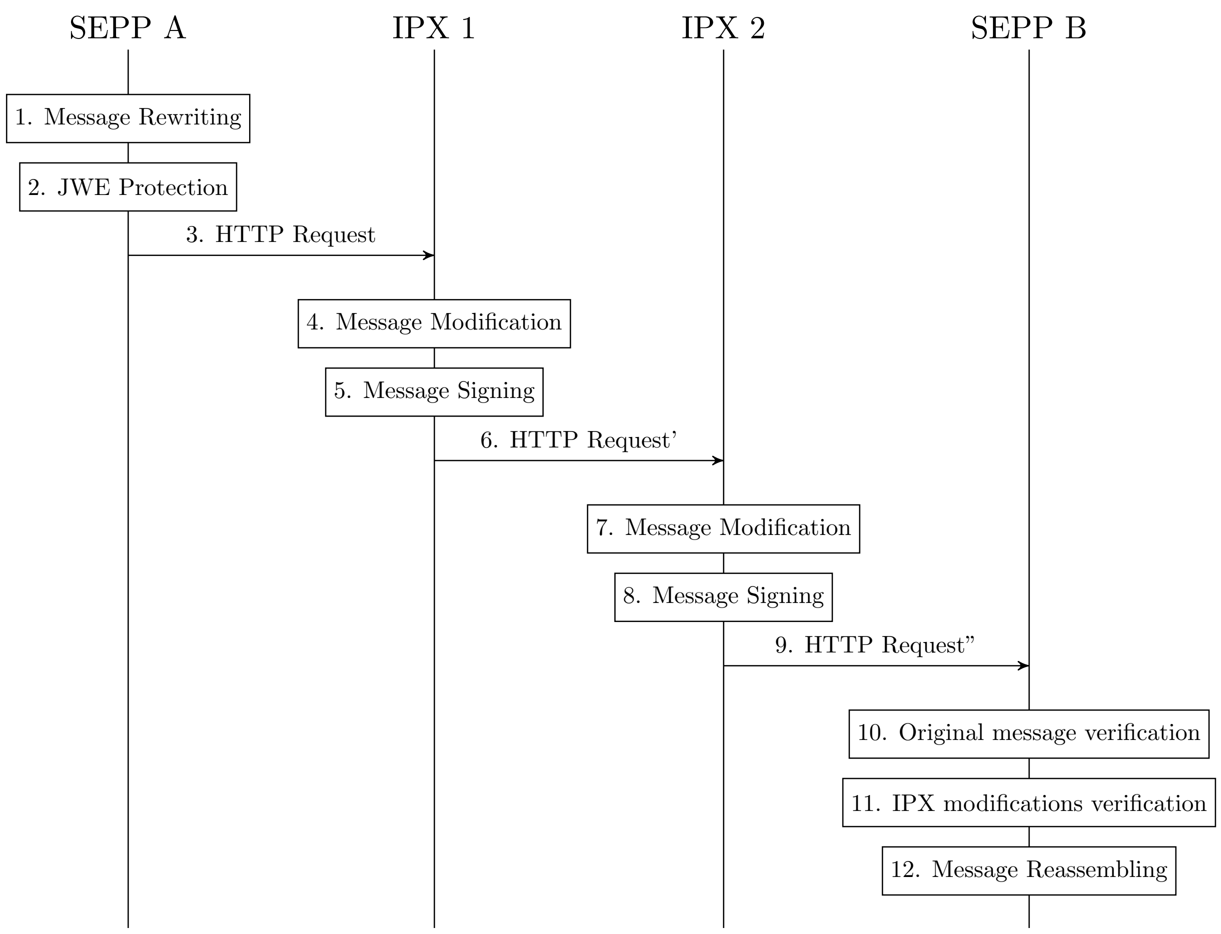


Figure 2.5: N32-f Sequence Diagram

1. After receiving a signaling message from an NF in its network, the sending SEPP rewrites the message into the JSON structures described previously.

2. On the resulting object, the sending SEPP applies JOSE protection as follows: The *dataToIntegrityProtectAndCipher* object serves as input to be ciphered, whereas the *dataToIntegrityProtect* object is used as AAD.

Security property 6: Information in the *dataToIntegrityProtect* object shall be integrity protected. (c.f. 3GPP 2019c, p. 138)

Security property 7: Information in the *dataToIntegrityProtectAndCipher* object shall be confidentiality and integrity protected. (c.f. 3GPP 2019c, p. 138)

3. The protected JSON object is embedded into the body of a new HTTP request and sent to the first IPX provider.

4. The first IPX provider in the path parses the *dataToIntegrityProtect* object and determines the necessary changes. It creates a new *modifiedDataToIntegrityProtect* object, containing a list of all modifications. If there are no modifications to be performed, the IPX provider shall write the empty list into the *modifiedDataToIntegrityProtect* object.

Security property 8: Intermediaries in the path shall not be able to remove message modifications previously added by IPX providers. (c.f. 3GPP 2019c, p. 142)

5. The JSON object created in the previous step is fed into the JWS algorithm, and appended to the original message together with the cryptographic signature over the whole message.

Security property 9: Message modifications by IPX providers shall be authenticated and integrity protected.

Security property 10: It shall not be possible to re-apply message modifications by IPX providers to N32-f messages at a later point in time (replay protection). (c.f. 3GPP 2019c, p. 142)

6. The modified message is embedded into the body of a new HTTP request and sent to the next hop.

7. Similar to step 4, the second IPX provider determines the changes necssary and adds appropriate JSON patches to the existing *modifiedDataToIntegrityProtect* object.

8. The second IPX provider signs the complete *modifiedDataToIntegrityProtect* object and appends the resulting signature to the message.

9. The modified message is embedded into the body of a new HTTP request and sent to the receiving SEPP.

10. After receving an N32-f message, a SEPP first verifies the integrity of the original message contents, i.e. the *dataToIntegrityProtectAndCipher* and *dataToIntegrityProtect* object by deciphering the confidential part using an AEAD decryption. This step only succeeds if both objects have not been modified.

11. Next, the SEPP verifies all IPX modifications by checking the contained JWS signatures and comparing the patch operations to the modification policies of both roaming partners.

Security property 11: Only IPX providers referenced in one of the SEPP operator's IPX security information lists shall be able to perform message modifications. (c.f. 3GPP 2019c, p. 143)

Security property 12: Only message modifications referenced in one of the SEPP operator's modification policies shall be possible. (c.f. 3GPP 2019c, p. 143)

12. Lastly, the signaling message is reassembled while applying all legitimate modifications successfully verified in the previous step. The resulting message is sent to its final destination.

2.2 Theory of Formal Verification

Due to the nature of formal protocol verification, not all of the security properties outlined in the previous section may be effectively verifiable using this method. The following section details what formal methods are generally able to prove and what not. Chapter 4 then goes on to discuss above properties in context and formalizes them, if possible.

2.2.1 Symbolic Model Checking and First-Order Logic

At its core, symbolic model checking describes the verification of a given system represented by a model of system states – a security protocol is essentially a distributed, concurrent system – against a temporal logic formula. The model can be, for example, a Finite-State Machine (see Alur and Yannakakis 1998), or a Binary Decision Diagram (see Burch et al. 1992), that captures the number of reachable states and transitions between them. System states are modeled in an abstract manner by symbols that represent information available to the participants at a given point in time. State transitions describe how the systems moves from one state into another, the precondition for it happening and the result of the transition. The temporal logic formula details the intended system requirements in a mathematical way. The act of model checking is the exploration of all system states, verifying whether or not the formula holds true for all possible scenarios. An issue is discovered if the provided model does not accurately match the formula – hence the term *model checking*. In practice, there exists a number of optimizations to make the task of exploring all system states more efficient than a brute-force search (see Etessami and Holzmann 2000). If these operations are performed by a computer, it is considered automatic model checking. Most model checkers, including the ones that are considered in this thesis, build on some subset of first-order logic, which shall be briefly introduced in the remainder of this section. These fundamental concepts are key to understand what kind of system properties are in scope of the following analysis.

First-Order Logic

First-Order Logic –also referred to as predicate logic– extends classical, propositional logic by the notion of predicates and quantifiers. Whereas the latter is solely concerned with logical statements and connectives, first-order logic is inherently connected to a domain of discourse or universe and the semantics associated with it. For example, propositional logic is only able to express binary relations using atomic formulas (or symbols), logical connectives (or functions), and formulas that combine symbols and functions according to the logic's particular grammar:

"If it is sunny (S), it is not raining (R)."

$$S \to \neg R$$

A first-order logic on the other hand is comprised of quantified symbols, functions, as well as propositions (or relations) between these elements. Functions and relations are not necessarily binary but can take any number of parameters greater than zero.

"All humans have mothers and John is human. Therefore, John has a mother."

$$(\forall x)(Hx \to Mx) \wedge (Hj) \to (Mj)$$

where $Hx :=$ x is human; $Mx :=$ x has a mother; $j :=$ constant 'John'

Or

"There exists x such that x is John's mother and x is female."

$$(\exists x)(Mxj \wedge Fx)$$

where $Mxy :=$ x is the mother of y; $Fx :=$ x is female; $j :=$ constant 'John'

First-order logic is defined relative to a *signature*, i.e. a set of symbols not inherent to classical logic, by a particular syntax and semantics. The syntax determines the set of well-formed formulas, whereas the semantics associates meaning to these formulas. This thesis follows the formal definition of Schöning (see Schöning 2008, pp. 41-47), the key points of which are extracted below.

The first-order logic syntax is comprised of variables, predicates, functions and quantifiers. *Variables* are denoted as x_i where $i = 1, 2, 3, \ldots$, *predicate symbol* as P_i^k, and a *function symbol* as f_i^k where $i = 1, 2, 3, \ldots$ and $k = 0, 1, 2, \ldots$. The subscript i is called the index whereas k represents the arity of a function. A function of arity 0 is called a *constant.* A *term* is defined as follows:

1. Each variable is a term.
2. If f is a function symbol with arity k, and if $t_1, \ldots, t_k$ are terms, then $f(t_l, \ldots, t_k)$ is a term.

Next, *formulas* are defined inductively as follows.

1. If P is a predicate symbol with arity k, and if $t_1, \ldots, t_k$ are terms, then $P(t_1, \ldots, t_k)$ is a formula.

2. For each formula $F, \neg F$ is a formula.

3. For all formulas F and G, $(F \wedge G)$ and $(F \vee G)$ are formulas.

4. If x is a variable and F is a formula, then $\exists x F$ and $\forall x F$ are formulas.

Formulas built in accordance to rule 1 are also called *atomic formulas.* If F is a formula, and F occurs as part of the the formula G, then F is called a *subformula* of G. All occurrences of a variable in a formula are distinguished into bound and free occurrences. An occurrence of the variable x in the formula F is bound if x occurs within a subformula of F of the form $\exists x G$ or $\forall x G$. A formula without occurrence of a free variable is called a *closed formula* or *sentence*. The symbol $\exists$ is called the *existential quantifier*. It denotes its argument holds true sometimes (read: „There exists $x, \ldots$"). The symbol $\forall$ is called the *universal quantifier*. It denotes its argument holds true everytime (read: „For all $x, \ldots$").

Each first-order logic is defined in relation to a domain of discourse that gives it meaning. A *structure* is a construct of the form $A = (U_A, I_A)$. U_A is an arbitrary, non-empty set called the *universe*. I_A is a mapping that maps

1. each k-ary predicate symbol P to a k-ary predicate on U_A (if I_A is defined on P).

2. each k-ary function symbol f to a k-ary function on U_A (if I_A is defined on f).

3. each variable x to an element of U_A (if I_A is defined on x).

Let F be a formula and $A = (U_A, I_A)$ be a structure. A is called *suitable* for F if I_A is defined for all predicate symbols, function symbols, and for all variables that occur free in F.

Let F be a formula and let $A = (U_A, I_A)$ be a suitable structure for F. For each term t occurring in F, its *value* under the structure A is denoted as $A(t)$ and defined inductively as follows.

1. If t is a variable ($i.e., t = x$), then we let $A(t) = x^A$.

2. If t has the form $t = f(t_1, \ldots, t_k)$ where $t_1, \ldots, t_k$ are terms and f is a function symbol of arity k, then we let $A(t) = f^A(A(t_1), \ldots, A(t_k))$.

1. If F has the form $F = P(t_1, \ldots, t_k)$ where $t_1, \ldots, t_k$ are terms and P is a predicate symbol of arity k, then

$$A(F) = \begin{cases} 1, & \text{if } (A(t_1), \ldots, A(t_k)) \in P^A \\ 0, & \text{otherwise} \end{cases}$$

2. If F has the form $F = \neg G$, then

$$A(F) = \begin{cases} 1, & \text{if } A(G) = 0 \\ 0, & \text{otherwise} \end{cases}$$

3. If F has the form $F = (G \wedge H)$, then

$$A(F) = \begin{cases} 1, & \text{if } A(G) = 1 \text{ and } A(H) = 1 \\ 0, & \text{otherwise} \end{cases}$$

4. If F has the form $F = (G \vee H)$, then

$$A(F) = \begin{cases} 1, & \text{if } A(G) = 1 \text{ or } A(H) = 1 \\ 0, & \text{otherwise} \end{cases}$$

5. If F has the form $F = \forall x G$, then

$$A(F) = \begin{cases} 1, & \text{if for all } u \in U_A, A_{[x/u]}(G) = 1 \\ 0, & \text{otherwise} \end{cases}$$

If for a formula F and a suitable structure A the term $A(F) = 1$ holds, then this is denoted by $A \models F$ (read A *models* F). If every suitable structure for F is a model for F, then this is denoted by $\models F$ (read F is *valid*). If there is at least one model for the formula F then F is called *satisfiable*, and otherwise *unsatisfiable*.

Translating this definition into the particular example at hand, first the PRINS protocol is to be transposed into formal a structure A („the model"). Secondly, a formula F has to be defined for each of the desired security properties in order to be able to validate it against A. If a model checker is able to prove that F is valid for A, the related property is considerd successfully verified. Given the nature of this approach to proving a system's correctness, model checking is only able to assess a particular set of properties. That is, only logical correctness can effectively be verified.

Safety Properties

Safety properties, particularly in the context of security protocols, can informally be described as the absence of something bad happening. A trivial example is secrecy: No unauthorized party is supposed to have access to sensitive data in the clear. These properties are specified over a particular order of states (or *traces*) the modeled system is going through and may be broken by a finite run that leads to an undesired state. Such cases, in which the reachability of a certain bad state falsifies the property are called *invariants*. Checking for this particular type of flaw is simply a matter of verifying whether a given furmula holds for all reachable states. More formally, invariants can be defined as follows (c.f. Baier and Katoen 2008, p. 107).

A property P_{inv} over A is an invariant if there is a first-order logic formula F over A such that

$$P_{inv} = \{A_0, A_1, A_2, \ldots \in (2^{AP})^\omega | \forall j \geq 0. A_j \models F\}$$

$(2^{AP})^\omega$ denotes the set of words arising from the infinite concatenation of words in the set of atomic propositions 2^{AP}. F is called an invariant condition of P_{inv} that holds true for every possible word, ruling out the reachability of a certain bad state.

In addition to invariants, there exist safety properties that are more involved than just the absence of certain undesired states. Instead, these may impose requirements on parts of the system's execution trace itself. Consider, for example, an arbitrary security protocol that would allow sending data prior to performing the cryptogrtaphic operations that ensure integrity or confidentiality. Even though this trivial case would likely already be discoverd during modeling phase, it examplifies that the order of subsequent states is relevant. That is, these type properties is not about guaranteeing the reachability of bad states, but bad path fragments in the execution trace. Some of the relevant features of security protocols that can be expressed expressed by this logic include confidentiality and authentication. Such safety properties are a generalization of invariants and are fullfilled if there is no execution trace that contains a finite prefix violating the premise.

According to Baier and Katoen, „a property P_{safe} over AP is called a safety property if for all words $\sigma \in (2^{AP})^{\omega} \in P_{safe}$ there exists a finite prefix $\widehat{\sigma}$ of σ such that

$$P_{safe} \cap \{\sigma' \in (2^{A})^{\omega} | \widehat{\sigma} \text{ is a finite prefix of } \sigma'\} = \emptyset$$

Any such finite word $\widehat{\sigma}$ is called a *bad prefix* for P_{safe}" (Baier and Katoen 2008, p. 112).

Liveliness Properties

Complementary to the abovementioned characteristics are properties that demand something good eventually always happening. Unlike safety properties, they do not impose requirements on finite traces of the system, but put conditions on the infinite behavior. A formal definition can be given as follows (c.f. Baier and Katoen 2008, p. 121).

Property P_{live} over A is a liveness property whenever $pref(P_{live}) = (2^{A})\star$. Thus, a liveness property (over A) is a property P such that each finite word can be extended to an infinite word that satisfies P. Stated differently, P is a liveness property if and only if for all finite words $w \in (2^{A})\star$ there exists an infinite word $\sigma \in (2^{A})^{\omega}$ satisfying $w\sigma \in P$.

Concrete examples for liveness properties are recovery, revocation, and the timeliness of certain actions.

Indistinguishability

The above definitions for both safety and liveness are closely linked to the concept of traces, i.e. individual system runs that satisfy or violate certain conditions. There exisits yet another type of properties that is not defined over individual traces, but the information an attacker is able to obtain by observing multiple instances of the system. This is commonly referred to as *indistinguishability* or *observational equivalence*. The formal definition by Baier and Katoen is provided below.

„Let TS_1 and TS_2 be two transition systems over A with state spaces S_1 and S_2, respectively. A binary relation $R \subseteq S_1 \times S_2$ is an observational bisimulation for (TS_1 , TS_2) if and only if the following conditions are satisfied:

(A) Every initial state of TS_1 is related to an initial state of TS_2, and vice versa. That is

$$\forall s_1 \in I_1 \exists s_2 \in I_2.(s_1, s_2) \in R \text{ and } \forall s_2 \in I_2 \exists s_1 \in I_1.(s_1, s_2) \in R$$

(B) For all $(s_1, s_2) \in R$ the following conditions (1), (2) and (3) hold:

(1) If $(s_1, s_2) \in R$ then $L_1(s_1) = L_2(s_2)$

(2) If $(s_1, s_2) \in R$ and $s'_1 \in Post(s_1)$ then there exists a path fragment $u_0 u_1 \ldots u_n$ such that $n \geq 0$ and $u_0 = s_2, (s'_1, u_n) \in R$ and, for some $m \leq n, L_2(u_0) = L_2(u_1) = \ldots = L_2(u_m)$ and $L_2(u_{m+1}) = L_2(u_{m+2}) = \ldots = L_2(u_n)$.

(3) If $(s_1, s_2) \in R$ and $s'_2 \in Post(s_1)$ then there exists a path fragment $u_0 u_1 \ldots u_n$ such that $n \geq 0$ and $u_0 = s_2, (u_n, s'_2) \in R$ and, for some $m \leq n, L_1(u_0) = L_1(u_1) = \ldots = L_1(u_m)$ and $L_1(u_{m+1}) = L_1(u_{m+2}) = \ldots = L_1(u_n)$.

TS_1 and TS_2 are called *observational equivalent*, denoted $TS_1 \approx_{obs} TS_2$, if there exists an observational bisimulation for (TS_1, TS_2)" (Baier and Katoen 2008, pp. 589-590).

2.2.2 Tamarin

The TAMARIN prover is a tool for automated formal verification in the symbol model. Initially developed by Meier and Schmidt at ETH Zürich (see Schmidt 2012, Meier 2013), its primary domain of application is the analysis of security protocols. TAMARIN enjoys notable popularity since its creation, with some of the protocols verified using the tool cited on its official website (c.f. Basin, Cremers, Dreier, et al. 2020), such as TLS 1.3 (Cremers et al. 2017), CoAP and MQTT (Kim et al. 2017), and 5G AKA (Basin, Dreier, Hirschi, et al. 2018).

The security protocol under test is specified in TAMARIN's custom syntax and internally represented as a Multiset Rewriting System (MRS). Security properties to be verified are written as two-sorted first-order logic formulas, i.e. formulas considering terms of two types: protocol messages and timepoints. It features built-in support for the aforementioned Dolev-Yao adversaries, but has also been extended to incorporate stronger attackers such as the Extended Canetti-Krawczyk (ECK) model (see LaMacchia, Lauter, and Mityagin 2007). In the following, an overview of the TAMARIN prover's key concepts is provided as well as a survey of its current feature set and limitations.

Terminology

State is modeled as a multiset, i.e. a set of sets, of facts. Facts are built of terms over a fact signature Σ_{Fact} and have the form $F(t_1, \ldots, t_k)$. They can either be linear or persistent. As Meier explains, „[l]inear facts model resources that can only be consumed once, whereas persistent facts model inexhaustible resources that can be consumed arbitrarily often" (Meier 2013, p. 76). The finite multiset of all facts represents the system state, including the adversary knowledge.

Rules are used to describe transitions between system states. Multiset rewriting rules are of the form $l \dashv a \vdash\!\!\rightarrow r$, where a multiset of facts l is replaced with a multiset of facts

r by action a. Such rule is applicable to a given state S if it contains the same linear and persistent facts as S. In order to transition into a successor state S', all linear facts are removed and facts generated by action a are added.
A protocol *execution* is defined by an alternating sequence of states and rules. The initial state S_0 is always the empty multiset.

$$e = [S_0, (l_1 \dashv a_1 \vdash\!\!\rightarrow r_1), S_1, \ldots, S_{k-1}, (l_k \dashv a_k \vdash\!\!\rightarrow r_k), S_k] \tag{2.1}$$

Based on above definition, the *trace* of a protocol execution is described by the sequence of actions to get from the initial to the final state.

$$trace(e) = [set(a_1), \ldots, set(a_k)] \tag{2.2}$$

This is key for describing certain safety properties on the basis of traces, as shown in the previous section.

Adversaries do not need to be defined manually, but are implicitly modeled within the multiset rewriting system of the protocol itself. In order to do so, TAMARIN uses the following fact symbols reserved for particular use cases.

- Out(x) denotes sending a message x via the public Dolev-Yao channel. Once this is done, the adversary can read, modify, or block any of the contained information.
- In(x) denotes receiving a message x via the public channel. Each incoming message originates with the Dolev-Yao adversary controling the channel.
- Fr(x) denotes a fresh name x that is unique and unguessable. This capability is available to the adversary as well and is required for random values or nonces, which are a common component of security protocols.
- K(x) denotes the adversary's knowledge of x, i.e. the contents of variable x are revealed.

Based on these unary fact symbols, TAMARIN defines a set of message deduction rules (c.f. Meier 2013, p. 78). These allow the adversary send and receive messages on the public channel, learn about freshly generated names and public constants, and apply arbitrary functions on the messages observed.

$$\mathsf{MD} = \left\{ \begin{array}{r l} \mathsf{Out(x)} & \dashv\vdash\!\!\rightarrow \mathsf{K(x)} \\ \mathsf{K(x)} & \dashv \mathsf{K(x)} \vdash\!\!\rightarrow \mathsf{In(x)} \\ \mathsf{Fr(x)} & \dashv\vdash\!\!\rightarrow \mathsf{K(x)} \\ [] & \dashv\vdash\!\!\rightarrow \mathsf{K(x{:}pub)} \\ (\mathsf{K}(x_1), \ldots, \mathsf{K}(x_k)) & \dashv\vdash\!\!\rightarrow \mathsf{K}(f(x_1, \ldots, x_k) | f \in \Sigma^k) \end{array} \right\}$$

Security properties

In TAMARIN security properties are denoted using two-sorted first-order logic formulas over message terms and timepoints. These so-called lemmas allow for the specification of various trace properties as well as some forms of observational equivalence.

As shown in section 2.2, one of the most common use case of trace properties as far as security protocols are concerned is specifying secrecy (recall that invariants are simply a special form of trace properties). Secrecy in TAMARIN can be expressed using the following lemma, stating that for any secret action x at timepoint i, there exsists no timepoint j at which the adversary has knowledge about x. The very last line is added to ensure satisfiability of the lemma in the face of a potential key compromise of an honest participant B at timepoint r. Beyond this basic secrecy property, the tool further offers support for perfect forward secrecy. (see The Tamarin Team 2019, pp. 71-72)

```
lemma secrecy:
    "All x #i.
        Secret(x) @i ==>
        not (Ex #j. K(x) @j)
            | (Ex B #r. Reveal(B) @r & Honest(B) @i)"
```

Listing 2.1: Tamarin secrecy query, according to The Tamarin Team 2019, p. 72

Trace properties are also used to specify authentication between protocol participants. As Lowe shows, there are several ways to define authentication that vary in stictness (see Lowe 1997). With the exception of Recentness, TAMARIN can model the following four types of authentication.

Aliveness mandates that if participant a completes a protocol run at timepoint i, there exists a participant b that has previously run the protocol. Intuitively, this is a rather weak premise and not exactly what most security protocols are supposed to achieve.

```
lemma aliveness:
    "All a b t #i.
        Commit(a,b,t) @i
        ==>  (Ex id #j. Create(b,id) @j)
            | (Ex C #r. Reveal(C) @r & Honest(C) @i)"
```

Listing 2.2: Tamarin aliveness query, according to The Tamarin Team 2019, p. 74

Weak agreement specifies that if participant a completes a protocol run at timepoint i, there exists a participant b that has previously run the protocol with a. While this is closer to the intuitive definition of authentication, this does not accurately model the PRINS requirement either, as both participants further need to agree a secret session key amongst each other.

```
lemma weak_agreement:
    "All a b t1 #i.
        Commit(a,b,t1) @i
        ==> (Ex t2 #j. Running(b,a,t2) @j)
            | (Ex C #r. Reveal(C) @r & Honest(C) @i)"
```

Listing 2.3: Tamarin weak agreement query, according to The Tamarin Team 2019, p. 75

Non-injective agreement mandates that if participant a completes a protocol run at timepoint i, there exists a participant b that has previously run the protocol with a and both participants agreed on a message term t. Note, that this does not guarantee both participants completing a protocol run the same number of times. That is, participant a could run the protocol three times with participant b, while for its part, the latter only recognizes one full protocol execution. Since this does not align with 3GPP's specification either, an even stricter form of authentication is required.

```
lemma noninjective_agreement:
  "All a b t #i.
    Commit(a,b,t) @i
    ==> (Ex #j. Running(b,a,t) @j)
      | (Ex C #r. Reveal(C) @r & Honest(C) @i)"
```

Listing 2.4: Tamarin non-injective agreement query, according to The Tamarin Team 2019, p. 75

Injective agreement specifies that if participant a completes a protocol run at timepoint i, there exists a participant b that has previously run the protocol with a and both participants agreed on a message term t. Furthermore, there exisits a one-to-one relationship between the number of protocol executions by a and b.

```
lemma injective_agreement:
  "All A B t #i.
    Commit(A,B,t) @i
    ==> (Ex #j. Running(B,A,t) @j
    & j < i
    & not (Ex A2 B2 #i2. Commit(A2,B2,t) @i2 & not (#i2 = #i)))
      | (Ex C #r. Reveal(C) @r & Honest(C) @i)"
```

Listing 2.5: Tamarin injective agreement query, according to The Tamarin Team 2019, p. 75

In addition to trace properties, Tamarin is also able to prove observational equivalence using the diff operator introduced by Basin, Dreier, and Sasse (see Basin, Dreier, and Sasse 2015). The underlying idea is to check whether two multiset rewriting system exhibit identical rules but vary only by their instanciation, i.e. by the valuation of their variables. If it would be possible to apply different rules in one of the systems that are not available in the other, the adversary would be able to distinguish the two. Internally, Tamarin duplicates the model and checks whether every rule execution in one instanciation has a corresponding rule execution in the other one. This type of check is particularly relevant for proving privacy properties, for example in e-voting protocols. (see The Tamarin Team 2019, pp. 57-64)

Featureset & Limitations

As established previously, Tamarin offers support for a range of trace properties that allow for the specification of common secrecy, authentication, and privacy goals. On top of that, it provides a number cryptographic primitives built-in that are commonly used in security protocols: hashes, signatures, symmetric and asymmetric encryption, diffie-hellman groups, bilinear groups, XOR operations, and multisets. The first three of these are of relevance to the PRINS protocol, as hash functions and symmetric encryption are key for defining JWE's AEAD encryption scheme and digital signatures are the core of JWS. Additional language features to optimize the formal verification include different channel properties (Dolev-Yao, authentic, confidential, and secure channels), various heuristics to optimize automated proof methods, and proof by induction.

In its current version, Tamarin does not support blind signatures and trapdoor commitments, i.e. non-subterm-convergent theories, although there are extensions that address specificly those (see Dreier et al. 2017). Since neither of these properties is required for the PRINS protocol, this does not hinder the verification in any way. When operated in observational equivalence mode, the tool is sound, but not complete. (see The Tamarin Team 2019, p. 107)

2.2.3 ProVerif

ProVerif is another example for a popular tool for symbolic model checking. Chiefly developed by Bruno Blanchet at INRIA Paris, it is one of the most mature model checking tools in existence, being actively maintained for almost 20 years. Like Tamarin, it has successfully been used to analyse a wide range of security protocols over the years. Examples cited on its official website (see Blanchet 2020) include the Signal protocol (Kobeissi, Bhargavan, and Blanchet 2017), the ARINC823 avionic protocols (Blanchet 2017), and the FOO, Lee, JCJ, and Belenios e-voting protocols (Hirschi and Cremers 2019).

Compared to the Tamarin prover, ProVerif approaches the problem of verifying security protocols distinctly different. Protocols are specified in the applied pi calculus (Abadi, Blanchet, and Fournet 2017), a variant of the pi calculus originally proposed by Milner (Milner 1999), a process algebra that is used to formally describe the behavior of security protocols. It extends the latter by components such as fresh variables and functions that model cryptographic primitives, to make it more suitable for this particular purpose.

Terminology

The **applied pi calculus** and process algebras in general are a formal method for describing the behavior of concurrent systems. As Baeten et al. point out: „A process algebra can be defined as any mathematical structure satisfying the axioms given for the basic operators" (Baeten, Beek, and Rooda 2007, p. 1). The term *process* is to be understood as an abstract element of this model. The fact that it is an *algebra* signals that algebraic rules apply. Accordingly, calculations on processes can be performed using the defined operators in a given algebra. In the applied pi calculus used in ProVerif, operators are refered to as expressions. Expressions perform computations on terms, representing data or messages that are exchanged between processes. Processes are built from terms and expressions. Processes are used to model protocol participants, the Dolev-Yao adversary, as well as the protocol under test itself.

Terms are built up from names, variables and constructors. While names represent atomic data, variables are a placeholder for other terms and can be replaced by such.

Constructors are functions with a certain arity used to build terms. All three kinds of terms are defined with a type. The default types defined by PROVERIF are channel, boolean, and bitstring, but the tool also supports the definition of custom types.

```
M,N ::=                      term
    x , y , z                variable
    a , b , c , k , s        name
    f(M_1,...,M_n)           constructor application
```

Listing 2.6: ProVerif term grammar, according to Blanchet et al. 2016, p. 13

Expressions evaluate so-called *may-fail* terms. Their evaluation, outlined in listing 2.7, yields a result that is either a term M or the special constant $fail$, representing a failed computation. This language feature allows for the modeling of complex data types as well as cryptographic primitives, such as encryption which constructs a ciphertext from atomic terms (key and plaintext) and decryption which resolves a *may-fail* term (ciphertext) into the original plaintext, unless it fails.

```
D ::=                        expression
    M                            term
    f(D_1,...,D_n)               constructor application
    g(D_1,...,D_n)               destructor application
    fail                         failure
```

Listing 2.7: ProVerif expression grammar, according to Blanchet et al. 2016, p. 13

Processes in PROVERIF largely coincide with processes in the original pi calculus (c.f. Blanchet et al. 2016, p. 18). They allow for the description of sending and receiving messages, parallel execution, creation of new names, evaluation of expressions, and conditionals. Listing 2.8 below shows the syntactical contructs available to do so.

```
P,Q ::=                      process
    0                            nil
    out(N,M);P                   output
    in(N,x:T);P                  input
    P|Q                          parallel composition
    !P                           replication
    new a:T;P                    restriction
    let x:T=D in P else Q        expression evaluation
    if M then P else Q           conditional
```

Listing 2.8: ProVerif process grammar, according to Blanchet et al. 2016, p. 13

Adversaries in PROVERIF do not need to be modeled explicitly. Instead, it has access to all constants, variables, constructors, destructors, and channels – unless specifically marked as private. During verification the tool attempts to mount attacks using all the available information. Similar to the TAMARIN prover, private information may be revealed when sent over the Dolev-Yao channel using the following keywords:

– out(c, x) denotes sending a message x on channel c. If c is a public channel, the adversary can read, modify, or block any of the contained information.

– in(c, x) denotes receiving a message x on channel c.

– new x denotes the generation of a fresh name x that is unique and unguessable.

Internally, ProVerif translates the applied pi calculus representation of a protocol into a set of **Horn clauses**. Horn clauses are a Turing-complete subset of first-order logic commonly used in logic programming and theorem proving. As stated in section 2.2.1, an atomic formula in predicate logic has the shape $P(t_1, \ldots, t_n)$. A *literal* L is either an atomic formula or the negation of one. A *clause* is a disjunction of literals, such as:

$$L_1 \vee L_2 \vee L_3 \vee \ldots \vee L_n \tag{2.3}$$

A *Horn clause* is a clause containing at most one positive literal. A Horn clause without any negative literals or variables is called a *fact*. Rather than in the disjunctive form displayed above, these clauses are often used to describe logical implication. In below example, L_3 follows from L_1 and L_2:

$$\neg L_1 \vee \neg L_2 \vee L_3 \Leftrightarrow \underbrace{L_3}_{\text{query}} \leftarrow \underbrace{L_1 \wedge L_2}_{\text{hypothesis}} \tag{2.4}$$

In logic programming and theorem proving, the latter is also called *goal clause*. Proving L_3 is equivalent to finding a solution to the right-hand side of the equation, i.e. its *hypothesis*. Alternatively, disproving it is a matter of finding a counterexample to the query. In its internal representation ProVerif effectively uses three types of Horn clauses (c.f. Blanchet 2013, p. 13):

– Clauses describing the computation abilities of the adversary, i.e. application of constructors, destructors and generation of fresh names

– Facts decribing the initial knowledge of the attacker, such as public keys

– A multiset of clauses describing the protocol, containing one set for each message

Security Properties

Security properties in ProVerif are specified using queries of a similar form as those introduced in equation 2.4. That way, the tool is capable of proving basic reachability properties, correspondence assertions, and observational equivalence. The first category allows for the verification of secrecy by checking whether the fact a given message M is exposed to the adversary is reachable under any circumstances. This can be denoted in two ways (c.f. Blanchet, Smyth, et al. 2020, p. 55):

```
query attacker(M)          // adversary cannot compute term M
query secret x             // adversary cannot compute bound variable x
```

Correspondence assertions on the other hand are the basis for verifying authentication properties. As described in the previous section, authentication can be defined in several different ways. ProVerif's user manual describes the verification of two of them - aliveness and injective agreement (see Blanchet, Smyth, et al. 2020, pp. 19-20). The keyword event can be used to annotate special points in a protocol that can then be checked for correspondence, as shown in the below code fragment. Effectively, below query is the equivalent of proving the aforementioned aliveness property, i.e. for every event receivedByA the event sentByB has been reached previously at least once. Injective agreements, requiring a one-to-one relationship between states can be expressed using the keyword inj-event. Additionally, weak agreements and non-injective agreements can be modeled by expanding the term M shared by both participants' events with further information.

```
event receivedByA(M); event sentByB(M);
query receivedByA(M) ==> sentByB(M)
```

ProVerif goes beyond these basic correspondence assertions, allowing for the use of conjunctions (&&), disjunctions (||), and nesting in the specification of hypotheses. Nested hypotheses are equivalent to a chain of logical dependencies of the form (c.f. Blanchet, Smyth, et al. 2020, p. 54):

```
event A; event B; event C;
A(x) ==> B(x) ==> C(x) || D(x) // A is preceded by B is preceded by C or D
```

Observational equivalence can take multiple forms, some of which can be verified by ProVerif (c.f. Blanchet, Smyth, et al. 2020, pp. 55-62). Firstly, it is able to prove strong secrecy, i.e. the indistinguishability of a change in secrets for the adversary, with the noninterf keyword. Secondly, resistence against offline guessing attacks can be verified with the weaksecret keyword by checking whether the adversary is able to distinguish correct from incorrect guesses of a secret when interacting with the protocol participants. Thirdly, observational equivalence of two processes instances implemented based on biprocess simulation similar to Tamarin, using the choice keyword. Lastly, ProVerif will also attempt to verify observational equivalence of two processes that differ in structure. However, merging different processes into a single biprocess requires at least a similar structure and does not always succeed.

Featureset & Limitations

ProVerif supports the verification of secrecy, authentication and privacy properties, similar to Tamarin. Unlike the latter, queries for weak agreement and non-injective agreement are not built-in and are not explicitly pointed out in the user manual. The tool offers more options for checking various types of observational equivalence. Whereas Tamarin only features this kind of analysis for a full protocol execution, ProVerif's choice, noninf, and weaksecret constructs allow for more granular checks. The cryptographic primitives supported out of the box are symmetric and asymmetric encryption, probabilistic encryption, hashes, signatures, diffie-hellman groups, zero-knowledge proofs, and trapdoor commitments (c.f. Blanchet, Smyth, et al. 2020, pp. 42-51). As pointed out before, the PRINS protocol makes use of symmetric encryption, hashes, and signatures. Thus, ProVerif does support modeling all necessary cryptographic operations. Similar to Tamarin, it also supports proof by induction and channel properties, albeit in a less granular fashion – channels can be declared as public Dolev-Yao channels or private ones not accessible by the adversary.

Additionally, ProVerif supports basic temporal constructs in the form of **phases** and **synchronization**. When using protocol phases, each has its own security properties and process communication across different phases is forbidden. If processes have not fully terminated when transitioning from one phase to another, they are discarded. (see Blanchet, Smyth, et al. 2020, pp. 41-42). Synchronization on the other hand allows for modeling a mandatory order of commands across processes. If one process has not executed certain commands prior a point of synchronization yet, it waits (potentially indefinitely) until these are executed. **Tables** can be used to model persistent storage of any of the participants that is not accessible by the adversary.

Since the problem of verifying security properties of a protocol with an unlimited message space and number of sessions is fundamentally undecidable, ProVerif may not always terminate. Furthermore, the fact that the tool employs certain approximations when translating protocols from the applied pi calculus into Horn clauses –mainly ignoring the number of repetitions of actions (c.f. Blanchet, Smyth, et al. 2020, p. 118)– can lead to the detection of false attacks that would not be feasible in real-world scenarios.

2.3 Summary

As outlined in section 2.1, PRINS is a non-trivial security protocol involving three active participants (initiating SEPP, responding SEPP, intermediary IPX providers), and two interdependent protocol channels. Its security goals evolve around confidentiality, integrity, replay protection, observational equivalenve, and authentication of message and patch sources. These properties do lend themselves to verification using formal methods.

The previous section 2.2 introduces the underlying theory and basic functionality of Tamarin and ProVerif. Before transposing the PRINS protocol specification into a model for formal verification, the question arises whether one is objectively better suited for this particular application.

Table 2.1 aims to capture modeling features of both Tamarin and ProVerif and to map them to the requirements of PRINS. AEAD encryption, which is an essential aspect of the N32-f protection, is assumed to be modeled using the built-in primitives for symmetric encryption and hashing. Depicted in this overview is only the „out-of-the-box" functionality of each respective tool. Based on that, both model checkers offer support for all cryptographic primitives necessary to model the PRINS protocol. As far as authentication is concerned, both Tamarin and ProVerif are able to address several different levels that cover the requirements of PRINS.

Different channel properties can ease modeling by encoding explicit assumptions about their security. Since TLS is not analyzed as part of this thesis (c.f. section 1.2), the N32-c channel is modeled as a secure channel. Both model checkers support this functionality.

Although the 3GPP specification does not explicitly spell out any observational equivalence requirements, it is to be assumed that the refresh of N32-f session keys should not be noticable to adversaries controlling the N32-f channel, given that this procedure is performed via the end-to-end protected N32-c channel. As noted before, ProVerif does

		PRINS		Tamarin	ProVerif
		N32-c	N32-f		
Cryptographic primitives	Symmetric encryption		●	✓	✓
	Asymmetric encryption			✓	✓
	Probabilistic encryption				✓
	Digital signatures		●	✓	✓
	Hashing		●	✓	✓
	Diffie-Hellman groups			✓	✓
	Zero-knowledge proofs				✓
	Trapdoor commitments				✓
	Bilinear groups			✓	
	XOR operations			✓	
Authentication types	Aliveness			✓	✓
	Weak agreement			✓	✓
	Non-injective agreement			✓	✓
	Injective agreement	●	●	✓	✓
Observational equivalence	Strong secrecy	●			✓
	Offline guessing				✓
	Equivalence of same processes			✓	✓
	Equivalence of similar processes				✓
Channel properties	Dolev-Yao		●	✓	✓
	Authenticated			✓	
	Confidential			✓	
	Secure	●		✓	✓
Additional features	Tables	○	○		✓
	Phases				✓
	Synchronization	○	○		✓

● *must-have* ○ *nice-to-have*

Table 2.1: Comparison of Tamarin and ProVerif features in relation to PRINS

offer finer control over the type of observational equivalence to be verified and proving strong secrecy of individual terms is a particular advantage over TAMARIN that is discernable based on these tools' documentation.

Furthermore, the additional features of PROVERIF can potentially help modeling the protocol more intuitively and closely aligned to the 3GPP specification. Its concept of tables can be utilized to recreate the N32-f context and the associated information stored by each of the SEPPs. Synchronization on the other hand is a suitable way to control the execution order of N32-c handshake and N32-f data transmission. As long as the handshake procedure described in section 2.1.4 is not finalized by both parties, there should be no N32-f communication. Both these constructs may not be strictly mandatory to accurately specify the PRINS protocol, but they certainly make it more straightforward and hence, contribute to avoiding mistakes during modeling. Therefore, from chapter 3 onwards, this thesis focuses on modeling and verifying PRINS using PROVERIF version 2.01.

Chapter 3

Modeling the PRINS Protocol

This chapter expains the ProVerif model of PRINS, design decisions taken during the modelling process, and its limitations. Section 3.1 describes the how the individual components of the protocol are represented by ProVerif constructs. Section 3.2 focusses on the assumptions and shortcuts that have been made due to the limitations of formal verification in general or ProVerif specifically.

The code fragments provided in in the remainder of this thesis follow a certain naming convetion. Variables are written in lowercase with underscores (e.g. var_a). Global constants are written in uppercase (e.g. CONST). Functions, constructors, destructors, and events are written in camel case (e.g. sendMsg).

3.1 Model Transposition

3.1.1 Supporting Constructs

As noted in the previous section, ProVerif features a basic type system. Aside from allowing for a closer representation of the actual protocol implementation, types also serve as guardrails while modeling the honest participants in that, for example, a constructor expecting an input of type bitstring cannot be called with a different type. However, it should be noted that types are ignored by the adversary during verification stage unless explicitly defined otherwise, allowing ProVerif to detect potential type flaw attacks (c.f. Blanchet, Smyth, et al. 2020, p. 38).

In addition to ProVerif's built-in data types, a number of custom types displayed in listing 3.1 are defined. Compound data structures, such as aad, prins, modprins, and ipxmod, are defined solely for convenience and improved readability so as to not pass each of their respective subcomponents individually. Other types are supposed to resemble data with specific characteristics. A Message Authentication Code (MAC), for example, is modeled by the type mac that cannot be transformed back into its original components. Data of type pubkey on the other hand, can only be derived from a related privkey.

```
type key.                          (* symmetric key *)
type mac.                          (* message authentication code *)
type http.                         (* http message *)
type privkey.                      (* asymmetric privat key *)
type pubkey.                        (* asymmetric public key *)
type prins.                         (* prins message *)
type modprins.                      (* modified prins message w/ patches *)
type aad.                           (* additional authenticated data *)
type ipxmod.                        (* ipx json patch *)
```

Listing 3.1: Custom type definitions

Listing 3.2 shows the defined channels used to transport messages between processes. By default, channels follow the Dolev-Yao model, meaning that all transferred information can be read, modified, intercepted or arbitrarily created by the adversary. Secure channels that the attacker has no access to can be modeled by explicitly annotating them as private.

Three public channels are defined in order to represent the logical N32-f message flow from the sending SEPP to the first IPX provider (nf32_a), from there to the next IPX node (nf32_i), and finally to the receiving SEPP (nf32_b) or vice versa. If all N32-f communication would be modeled using a single channel, the verification tool would have no information about the intended message flow involing the intermediary nodes. In such a case, a message sent by one SEPP could be received directly by another SEPP without traversing any of the IPX providers. Beyond that, separate channels also alleviate the need to encode full-featured routing information in every single message, detailed further in section 3.2.

In contrast to N32-f, N32-c communication utilizes private channels. Since communication follows a client-server model, each SEPP maintains a dedicated channel for sending messages. Logically, these represent end-to-end channels between two communicating SEPPs guaranteeing both confidentiality and integrity. As highlighted in section 1.2, proving security of established, well-tested protocols is considered out of scope. Since the protection of N32-c traffic completely relies on TLS, the protocol is assumed to perfectly secure in that regard.

The remaining channels transport messages within the SEPPs themselves or their respective PLMNs and thus, are also considered secure. Channels err_a and err_b are used internally to the two peers, SEPP A and SEPP B, to pass information about errors in N32-f messages from the receiving N32-f processes to the those handling error signaling via N32-c. Channels plmn_a and plmn_b connect the NF in each of the networks to its local SEPP.

```
free n32f_a: channel.               (* Channel between SEPP A and IPX A *)
free n32f_i: channel.               (* Channel between IPX Providers *)
free n32f_b: channel.               (* Channel between IPX B and SEPP B *)
free n32c_a: channel [private].     (* Channel between SEPP A and SEPP B *)
free n32c_b: channel [private].     (* Channel between SEPP A and SEPP B *)
free plmn_a: channel [private].     (* PLMN A internal channel *)
free plmn_b: channel [private].     (* PLMN B internal channel *)
free err_a: channel [private].      (* Internal error signaling Sepp A *)
free err_b: channel [private].      (* Internal error signaling Sepp B *)
```

Listing 3.2: Custom channel declarations

Listing 3.3 provides an overview of global constants used by the protocol model. These are utilized at different points of the process execution to represent fixed, known values or flags. An empty JSON patch written by IPX providers is modeled by the EMPTY constant (see listing 3.16). SEND and RECV flags are utilized, for example, during derivation of the N32-f keys to indicate whether or not the SEPP initiated the N32-c handshake (see listing 3.8 and 3.9). SUCC and FAIL are constants representing the status code for success or failure in error signaling procedures (see listing 3.10 and 3.11) Lastly, REQ and RES indicate the type of HTTP message being exchanged between the two NFs communicating via the inter-operator interface.

```
const EMPTY: ipxmod.                   (* the empty ipx patch *)
const SEND: bitstring.                 (* sending flag *)
const RECV: bitstring.                 (* receiving flag *)
const SUCC: bitstring.                 (* success flag *)
const FAIL: bitstring.                 (* failure flag *)
const REQ: bitstring.                  (* HTTP Request *)
const RES: bitstring.                  (* HTTP Response *)
```

Listing 3.3: Global constant declarations

ProVerif's tables are used to model each SEPP's local storage for N32-f context information. Each SEPP has to keep track of several attributes, listed in section 2.1, in order to correlate N32-f messages to a previously established session and check the validity of incoming messages. The created model does not consider all of these information, abstracting away the following attributes cited in the 3GPP specification: (1) The remote SEPP ID, part of the N32-f peer information, is not considered. In real-world implementations, this attribute helps peers to distinguish multiple SEPP instances in a foreign PLMN. The model assumes there is only a single instance in each network. (2) The N32-f context information, which contains information about the validity of session keys and whether or not TLS shall be used to transport N32-f messages. Firstly, the session key lifetime cannot be verified using formal methods, due to the lack of appropriate temporal constructs. Secondly, it is assumed PRINS is exclusively used to carry N32-f messages.

```
table storeSeppA(
    bitstring,                         (* SEPP B PLMN ID *)
    bitstring,                         (* SEPP B Address *)
    bitstring,                         (* N32-f Context ID *)
    nat,                               (* N32-f Message Counter *)
    key,                               (* parallel request key *)
    key,                               (* parallel response key *)
    key,                               (* reverse request key *)
    key,                               (* reverse response key *)
    bitstring,                         (* parallel request IV *)
    bitstring,                         (* parallel response IV *)
    bitstring,                         (* reverse request IV *)
    bitstring,                         (* reverse response IV *)
    bitstring,                         (* JWE Ciphersuite *)
    bitstring,                         (* Data-type encryption policy *)
    bitstring,                         (* SEPP A modification policy *)
    bitstring,                         (* SEPP B modification policy *)
    bitstring,                         (* IPX A id *)
    pubkey,                            (* IPX A public key *)
    bitstring,                         (* IPX B id *)
```

```
     pubkey).                          (* IPX B public key *)
```

Listing 3.4: Declaration of local SEPP storage using tables

Data constructors are used to model type transformations in ProVerif. Listing 3.5 shows an example that combines the information elements added by intermediate IPX providers when applying message modifications. This particular constructor takes four information elements of type bitstring, bitstring, mac, and bitstring and produces a single ipxmod element. Fundamentally, this follows the same syntax as ordinary ProVerif constructors. However, the [data] annotation has the effect that the adversary is able to deconstruct the resulting data type into its original subcomponents at any time (c.f. Blanchet, Smyth, et al. 2020, p. 37), even without a related deconstructor explicitly defined.

```
fun jsonPatch(bitstring,               (* Operations *)
     bitstring,                        (* IPX Id *)
     mac,                              (* JWE Tag *)
     bitstring): ipxmod [data].        (* JWS Signature *)
```

Listing 3.5: Definition of custom data constructor

Listing 3.6 shows an example of custom functions used in the PRINS model. By default, they are available to all protocol participants, including the adversary. Below functions for the derivation of N32-f context ID, master key, session key, AES initialization vector and nonce as well as the application of JSON patches to an incoming N32-f message are one-way functions. Since none of them defines a related deconstructor, the original function inputs cannot be derived from its output.

The function deriveMasterKey is uniquely defined as a private function due to a limitation of the created model. According to the 3GPP specification, the N32-f master key is derived from the N32-c session using TLS keying material exporters (3GPP 2019c, p. 132). Since the TLS protocol is not modeled in detail, the deriveMasterKey function is defined as a substitute not available to the Dolev-Yao adversary. Section 3.2 further describes the benefits and drawbacks of modeling the key derivation in this way.

```
fun deriveContextId(bitstring, bitstring): bitstring.
fun deriveSessionKey(bitstring, bitstring, key): key.
fun deriveIV(bitstring, bitstring, key): bitstring.
fun deriveNonce(bitstring, nat): bitstring.
fun applyPatches(bitstring, bitstring, bitstring): bitstring.
fun deriveMasterKey(bitstring): key [private].
```

Listing 3.6: Definition of custom one-way functions

In contrast to the functions displayed above, there are other constructors defined with one or more related deconstructors, reverting a particular operation. This is examplified by the sign function in listing 3.7. Given two inputs of type bitstring and privkey, the function creates a new bitstring, resembling a digital signature. The related deconstructor checkSign takes two arguments of type bitstring and pubkey and returns the original message. This operation only succeeds with the public key belonging to the private key that was previously used for signing the message, i.e. if the signature is valid. The second deconstructor getMess depicted in the listing below is provided to allow the adversary to retrieve signed message contents without possessing the public key for validation, as suggested by the official ProVerif user's manual (c.f. Blanchet, Smyth, et al. 2020, p. 14).

```
fun pk(privkey): pubkey.
fun sign(bitstring, privkey): bitstring.
reduc forall m: bitstring, k: privkey;
    checkSign(sign(m, k), pk(k)) = m.
reduc forall m: bitstring, k: privkey;
    getMess(sign(m, k)) = m.
```

Listing 3.7: Definition of custom constructors and destructors

3.1.2 N32-c Processes

According to the specification, N32-c communication comprises three different procedures: Key agreement, parameter exchange (e.g. protection policies or IPX provider information), and error handling (c.f. 3GPP 2019c, p. 128). The model created as part of this work combines the first two into a single procedure for initial session establishment and considers a separate error signaling procedure. Each of them is defined by a sending and a receiving process. Further, each process is modeled separately for SEPP A and SEPP B, since all processes access local storage, represented by a distinct table for each SEPP (see listing 3.4).

Listing 3.8 shows the sending process of the N32-c handshake for SEPP A. Initially, a fresh N32-f pre-context ID is created and sent towards the peer SEPP the N32-c channel, together with the configured cipher suites. The receipt of the second message in lines 457-459 shows that the same information is expected from the other party as well, containing the cipher suite agreed by both endpoints. This input operation validates whether the destination address maches the local SEPP address and only proceeds to the creation of a final N32-f context ID from the two precontext IDs if this condition is met. This address check for incoming messages is necessary to ensure correct execution of the protocol flow. Section 3.2 explains this requirement in detail and what impact it has on the verification result.

Subsequently, the SEPPs exchange encryption policies and modification policies. The information sent out as part of message 3 is accompanied by the destination SEPP address and the N32-f context ID agreed during the previous step. The process only continues past the receipt of message 4 if the incoming message contains the correct destination address, N32-f context ID, and an encryption policy that matches its own. Recall that the 3GPP specification assumes these protection policies to be agreed by the SEPP operators in advance (see section 2.1). Following a successful protecion policy exchange, messages 5 and 6 are used to share the IPX security information lists, i.e. an identifier of each SEPP's directly connected IPX provider and its public key.

Subsequently, the master key is derived using the above-mentioned constructor deriveMasterKey. Based on this master key, the complete hierarchy of session keys and initialization vectors is derived as outlined in section 2.1. Since the protocol distinguishes session keys and initialization vectors to be used based on the SEPP's role in the N32-c communication as well as the type of HTTP message transported, the functions deriveSessionKey and deriveIV consume a flag indicating whether the SEPP initiated the N32-c session (SEND) or not (RECV) and a flag indicating its use for HTTP requests or responses, in addition to the master key itself. Lastly, the agreed session parameters are stored in the SEPP's local storage table.

```
let N32cSendHandshakeSeppA(sepp_a_plmn: bitstring, sepp_a_addr: bitstring,
    sepp_b_plmn: bitstring, sepp_b_addr: bitstring, ciphers_a: bitstring,
    encp_a: bitstring, modp_a: bitstring, ipx_a_id: bitstring,
    ipx_a_key: pubkey)=

    (* derive N32-f context id & validate ciphersuites *)
    new n32f_pid: bitstring;
    (* msg 1: Sec_Param_Ex_Req *)
    out(n32c, (sepp_b_addr, ciphers_a, n32f_pid));
    (* msg 2: Sec_Param_Ex_Resp *)
    in(n32c, (
        =sepp_a_addr, ciphers_b: bitstring, n32f_pid_b: bitstring
    ));
    let n32f_cid = deriveContextId(n32f_pid, n32f_pid_b) in
    event sendN32fContext(sepp_a_addr, n32f_pid, n32f_pid_b, n32f_cid);

    (* verify matching encryption policies *)
    (* msg 3: Sec_Param_Ex_Req *)
    out(n32c, (sepp_b_addr, n32f_cid, encp_a, modp_a));
    (* msg 4: Sec_Param_Ex_Resp *)
    in(n32c, (
        =sepp_a_addr, =n32f_cid, =encp_a, modp_b: bitstring
    ));

    (* exchange ipx information *)
    (* msg 5: Sec_Param_Ex_Req *)
    out(n32c, (sepp_b_addr, n32f_cid, ipx_a_id, ipx_a_key));
    (* msg 6: Sec_Param_Ex_Resp *)
    in(n32c, (
        =sepp_a_addr, =n32f_cid, ipx_b_id: bitstring, ipx_b_key: pubkey
    ));
    (* bidirectional n32c channel remains open *)

    (* key derivation *)
    let msg_cnt_a = 0 in
    let master_key_a = deriveMasterKey(n32f_cid) in
    event sendMasterKey(sepp_a_addr, n32f_cid, master_key_a);
    let par_req_key_a = deriveSessionKey(SEND, REQ, master_key_a) in
    let rev_req_key_a = deriveSessionKey(RECV, REQ, master_key_a) in
    let par_res_key_a = deriveSessionKey(SEND, RES, master_key_a) in
    let rev_res_key_a = deriveSessionKey(RECV, RES, master_key_a) in
    let par_req_iv_a = deriveIV(SEND, REQ, master_key_a) in
    let rev_req_iv_a = deriveIV(RECV, REQ, master_key_a) in
    let par_res_iv_a = deriveIV(SEND, RES, master_key_a) in
    let rev_res_iv_a = deriveIV(RECV, RES, master_key_a) in

    (* store session parameters *)
    insert storeSeppA(
        sepp_b_plmn, sepp_b_addr, n32f_cid, msg_cnt_a, par_req_key_a,
        par_res_key_a, rev_req_key_a, rev_res_key_a, par_req_iv_a,
        par_res_iv_a, rev_req_iv_a, rev_res_iv_a, ciphers_a, encp_a,
        modp_a, modp_b, ipx_a_id, ipx_a_key, ipx_b_id, ipx_b_key
    ).
```

Listing 3.8: Definition of the sending N32-c handshake process

The receiving N32-c handshake process in listing 3.9 is structured similarly to the sending one, with a few key distinctions. Firstly, the SEPP creates a fresh N32-f precontext ID. Secondly, message one containing the destination SEPP address, a list of cipher suites, and a N32-f precontext ID are received from the SEPP initiating the communication. If the address and cipher suites match the receiving SEPP's locally configured parameters, it responds with its own N32-f precontext ID and the agreed cipher suite in message 2. The final N32-f context ID is derived in the same way as by the sending SEPP.

Subsequently, the protecion policies are exchanged in messages 3 and 4. As is the case in the sending process, the receiving party of the N32-c handshake only proceeds if the N32-f context ID and the configured encryption policies in both SEPPs are the same. This step is again followed by the exchange of information related to the involved IPX providers.

The parameter exchanges are followed by the derivation of master key, session keys and initialization vectors. The only difference to the sending N32-c handshake process is the flags used for creation of the session keys. In the listing below, the parallel session key and IV are derived with the constant parameter RECV, since the SEPP is on the receiving end of the initial N32-c handshake. In contrast, the function deriveSessionKey uses parameter SEND for the reverse session key in initialization vector. Lastly, all session parameters are stored in the SEPP's local storage.

```
let N32cRecvHandshakeSeppA(sepp_a_plmn: bitstring, sepp_a_addr: bitstring,
    sepp_b_plmn: bitstring, sepp_b_addr: bitstring, ciphers_a: bitstring,
    encp_a: bitstring, modp_a: bitstring, ipx_a_id: bitstring,
    ipx_a_key: pubkey)=

    (* derive N32-f context id & validate ciphersuites *)
    new n32f_pid: bitstring;
    (* msg 1: Sec_Param_Ex_Req *)
    in(n32c, (
        =sepp_a_addr, =ciphers_a, n32f_pid_b: bitstring
    ));
    (* msg 2: Sec_Param_Ex_Resp *)
    out(n32c, (sepp_b_addr, ciphers_a, n32f_pid));
    let n32f_cid = deriveContextId(n32f_pid_b, n32f_pid) in
    event recvN32fContext(sepp_a_addr, n32f_pid_b, n32f_pid, n32f_cid);

    (* verify matching encryption policies *)
    (* msg 3: Sec_Param_Ex_Req *)
    in(n32c, (
        =sepp_a_addr, =n32f_cid, =encp_a, modp_b: bitstring
    ));
    (* msg 4: Sec_Param_Ex_Resp *)
    out(n32c, (sepp_b_addr, n32f_cid, encp_a, modp_a));

    (* exchange ipx information *)
    (* msg 5: Sec_Param_Ex_Req *)
    in(n32c, (
        =sepp_a_addr, =n32f_cid, ipx_b_id: bitstring, ipx_b_key: pubkey
    ));
    (* msg 6: Sec_Param_Ex_Resp *)
    out(n32c, (sepp_b_addr, n32f_cid, ipx_a_id, ipx_a_key));
    (* bidirectional n32c channel remains open *)

    (* key derivation *)
    let msg_cnt_a = 0 in
    let master_key_a = deriveMasterKey(n32f_cid) in
    event recvMasterKey(sepp_a_addr, n32f_cid, master_key_a);
    let par_req_key_a = deriveSessionKey(RECV, REQ, master_key_a) in
    let rev_req_key_a = deriveSessionKey(SEND, REQ, master_key_a) in
    let par_res_key_a = deriveSessionKey(RECV, RES, master_key_a) in
    let rev_res_key_a = deriveSessionKey(SEND, RES, master_key_a) in
    let par_req_iv_a = deriveIV(RECV, REQ, master_key_a) in
    let rev_req_iv_a = deriveIV(SEND, REQ, master_key_a) in
    let par_res_iv_a = deriveIV(RECV, RES, master_key_a) in
    let rev_res_iv_a = deriveIV(SEND, RES, master_key_a) in

    (* store session parameters *)
    insert storeSeppA(
        sepp_b_plmn, sepp_b_addr, n32f_cid, msg_cnt_a, par_req_key_a,
        par_res_key_a, rev_req_key_a, rev_res_key_a, par_req_iv_a,
        par_res_iv_a, rev_req_iv_a, rev_res_iv_a, ciphers_a, encp_a,
        modp_a, modp_b, ipx_a_id, ipx_a_key, ipx_b_id, ipx_b_key
    ).
```

Listing 3.9: Definition of the receiving N32-c handshake process

Listing 3.10 shows the sending SEPP's part of error signaling process carried out over N32-c, the details of which are described not in the security specification TS 33.501, but TS 29.573 (c.f. 3GPP 2019b, p. 18).

After detecting an error in an incoming N32-f message, modeled by receiving a message containing on the internal channel err_a, the SEPP retrieves the related N32-f context from its local storage. Based on this information, the process initiates an error message over N32-c, containing the destination SEPP's address, the N32-f context ID, and the ID of the flawed message received.

The SEPP then awaits a response from its peer, confirming the receipt of said error message. The related input operation validates the correct SEPP address, matching N32-f context ID and message ID in order to ensure different error messages are not mixed up. How the contents of this response are determined by the SEPP responding to such error signaling is shown in the next listing 3.11.

```
let N32cSendErrorSeppA(sepp_a_addr: bitstring)=
    in(err_a, (n32f_context': bitstring, msg_id: nat));
    get storeSeppA(
        sepp_b_plmn, sepp_b_addr, n32f_context, msg_cnt_a, par_req_key_a,
        par_res_key_a, rev_req_key_a, rev_res_key_a, par_req_iv_a,
        par_res_iv_a, rev_req_iv_a, rev_res_iv_a, ciphers_a, encp_a,
        modp_a, modp_b, ipx_a_id, ipx_a_key, ipx_b_id, ipx_b_key
    ) suchthat n32f_context' = n32f_context in

    (* send error notification *)
    out(n32c, (sepp_b_addr, n32f_context, msg_id));

    (* receive confirmation *)
    in(n32c, (
        =sepp_a_addr, =n32f_context, =msg_id, status_code: bitstring
    ));
    event sendErrorSeppA(n32f_context, msg_id, status_code).
```

Listing 3.10: Definition of the sending error signaling process

Listing 3.11 contains the receiving part of the error signaling procedure. After receiving an error signaling message from its peer on N32-c, the SEPP retrieves the locally stored session parameters, based on the N32-f context ID.

The SEPP performs a validation of the received message ID, checking whether it is lower than or equal to the locally maintained message counter. This ensures a previous message with the same ID has indeed been sent previously. A confirmation message including status code is signaled back to the peer SEPP over N32-c. If above-mentioned check is successful, this status code is the constant SUCC, else FAIL.

```
let N32cRecvErrorSeppA(sepp_a_addr: bitstring)=
    in(n32c, (
        =sepp_a_addr, n32f_context': bitstring, msg_id: nat
    ));
    get storeSeppA(
        sepp_b_plmn, sepp_b_addr, n32f_context, msg_cnt_a, par_req_key_a,
        par_res_key_a, rev_req_key_a, rev_res_key_a, par_req_iv_a,
        par_res_iv_a, rev_req_iv_a, rev_res_iv_a, ciphers_a, encp_a,
        modp_a, modp_b, ipx_a_id, ipx_a_key, ipx_b_id, ipx_b_key
    ) suchthat n32f_context' = n32f_context in

    (* send confirmation *)
    if msg_id <= msg_cnt_a then
    (
        out(n32c, (sepp_b_addr, n32f_context, msg_id, SUCC));
        event recvErrorSeppA(n32f_context, msg_id, SUCC)
    )
    else
    (
        out(n32c, (sepp_b_addr, n32f_context, msg_id, FAIL));
        event recvErrorSeppA(n32f_context, msg_id, FAIL)
    ).
```

Listing 3.11: Definition of the receiving error signaling process

3.1.3 N32-f Processes

Prior to any inter-operator signaling, an NF has to initiate an HTTP request towards a recipient in a foreign network. Listing 3.12 shows the process of an NF in PLMN A creating a fresh message, compiling it into an HTTP request including source address, destination address, and message type (REQ) before sending it to its local SEPP over the plmn_a channel. The actual information sent to the SEPP is a tuple of the message direction (SEND) and the HTTP message itself. While the former is not necessary in real-world implementations, it prevents messages being erroneously consumed from the NF response process displayed in the next listing.

```
let NfARequest(nf_a_addr: bitstring, nf_b_addr: bitstring)=
    new msg_body: bitstring;
    let http_message =
        createHttp(nf_a_addr, nf_b_addr, REQ, msg_body) in
    event sendHttpMsgNfA(REQ, msg_body);
    out(plmn_a, (SEND, http_message)).
```

Listing 3.12: Definition of the requesting NF process

The NF response process shown in listing 3.13 expects an incoming HTTP message. The equality sign before the constant RECV is a shorthand for the validation of the message direction parameter. In this case, only tuples consisting of the constant RECV and a variable of type http will be consumed by the process.

Subsequently, the HTTP message is deconstructed into its original components. The action of the NF varies depending on the contained message type. If the message received is a request, a response is being sent that contains the same message body. Otherwise, only an event is triggered indicating the receipt of an HTTP response.

```
let NfAResponse(nf_a_addr: bitstring, nf_b_addr: bitstring)=
    in(plmn_a, (=RECV, msg: http));
    let createHttp(nf_b_addr: bitstring,
        =nf_a_addr,
        msg_type: bitstring,
        msg_body: bitstring) = msg in
    if msg_type = REQ then
    (
        event sendHttpMsgNfA(RES, msg_body);
        out(plmn_a, (SEND, createHttp(nf_a_addr, nf_b_addr, RES,msg_body)))
    )
    else event recvHttpMsgNfA(RES, msg_body).
```

Listing 3.13: Definition of the responding NF process

Listing 3.14 shows the sending N32-f process, i.e. the process in which signaling data is actually transferred. Initially, an outbound HTTP message is consumed on the channel between local PLMN channel, marked by the SEND flag in the input operation. After retrieving a N32-f session context, said HTTP message is deconstructed into its original contents using the createHttp data constructor. This yields the message body which is used in combination with the stored encryption policies to determine confidential and non-confidential message parts.

Next, the message is assigned a unique identifier by incrementing the locally maintained message counter. Together with other nonconfidential message parts serving as additional authenticated data, it is combined into a single structure using the constructor combineAAD. Prior to the encryption operation, the correct session key and nonce are determined based on the type of the HTTP message transported (REQ or RES).

The encryption function aeadEncrypt uses the session key, the nonce, the confidential parts of the signaling message, and the AAD to produce the encrypted message content (payload) and a JWE tag required for authentication. These information are supplemented by the destination address of the receiving SEPP as shown in figure 2.3 and then sent on the public N32-f channel. Lastly, the locally stored N32-f context is updated to reflect the incremented message counter.

```
let N32fSendSeppA()=
    in(plmn_a, (=SEND, http_message: http));
    get storeSeppA(
        sepp_b_plmn, sepp_b_addr, n32f_context, msg_cnt_a, par_req_key_a,
        par_res_key_a, rev_req_key_a, rev_res_key_a, par_req_iv_a,
        par_res_iv_a, rev_req_iv_a, rev_res_iv_a, ciphers_a, encp_a,
        modp_a, modp_b, ipx_a_id, ipx_a_key, ipx_b_id, ipx_b_key
    ) in

    (* rewrite http message *)
    let createHttp(source_addr: bitstring,
        destination_addr: bitstring,
        msg_type: bitstring,
        msg_body: bitstring) = http_message in
    event recvHttpMsgSeppA(msg_type, msg_body);
    let conf: bitstring = getConf(msg_body, encp_a) in
    let nonconf: bitstring = getNonconf(msg_body, encp_a) in

    let msg_id: nat = msg_cnt_a + 1 in
    let associated_data = combineAAD(
        source_addr, destination_addr, msg_type,
        nonconf, msg_id, ipx_a_id, n32f_context
    ) in

    (* determine session key & nonce *)
    let key = (if msg_type = REQ then
        par_req_key_a else
        par_res_key_a) in
    let nonce = (if msg_type = REQ then
        deriveNonce(par_req_iv_a, msg_id) else
        deriveNonce(par_res_iv_a, msg_id)) in

    let (payload: bitstring, jwe_tag: mac) = aeadEncrypt(
        key,
        nonce,
        conf,
        associated_data
    ) in
    event sendN32fMsgSeppA(n32f_context, msg_id, conf, nonconf);
    out(n32f_a, (sepp_b_addr, prins'(
        ciphers_a, nonce, associated_data, payload, jwe_tag
    )));
    insert storeSeppA(
        sepp_b_plmn, sepp_b_addr, n32f_context, msg_cnt_a, par_req_key_a,
        par_res_key_a, rev_req_key_a, rev_res_key_a, par_req_iv_a,
        par_res_iv_a, rev_req_iv_a, rev_res_iv_a, ciphers_a, encp_a,
        modp_a, modp_b, ipx_a_id, ipx_a_key, ipx_b_id, ipx_b_key
    ).
```

Listing 3.14: Definition of the sending N32-f signaling transmission process

The receiving part of N32-f signaling is modeled by the process in listing 3.15. The SEPP initially awaits a modified PRINS message with a destination address that matches its own. Note, that even if none of the intermediate IPX providers is required to make changes to a N32-f message, they are still be expected to each create an empty JSON patch and sign it - hence the expected data type modprins.

Next, the received message is decomposed into its original components written by the sending SEPP and the two added patches using the data constructor prins''. The AAD object is further broken down to access non-confidential parts of the original message as well as the N32-f message ID, the identity of the authorized IPX, and the related N32-f context. The latter is required right afterwards to look-up the matching session context in local storage.

Similarly to the sending process, the receiving side also has to determine the session key and nonce based on the HTTP message type, before performing the decryption. The following destructor aeadDecrypt, requires these two parameter as well as the encrypted payload, the JWE tag, and the AAD object in order to perform the decryption. If successful, it yields the confidential parts of the original N32-f signaling message.

Afterwards, the first JSON patch is validated by checking whether its origin is the same as specified by the sending SEPP in parameter auth_ipx_id, by ensuring the operations are part of the modification policy, and by validating the contained JWE tag is the same as the one in the original message. All three checks are performed during decomposition of the ipxmod object by matching them against the expected values. Subsequently, the validity of the JWS object is checked using the constructor validPrinsSign. If all four checks pass successfully, the patch is considered valid, else invalid, and the relevant information is sent on the previously mentioned error signaling channels. The modification of the patch by the second IPX provider is omitted in the listing below for brevity. Validation happens in the exact same way with one minor deviation: Instead of validating the source of the JSON patch against the authorized IPX identifier specified by the sending SEPP, it is compared against the identifier of the second IPX provider on the path, which is locally stored in the N32-f context.

Lastly, if both JSON patch validations succeed, the message as a whole is considered valid. In this case, the constructor applyPatches is used to incorporate the changes added by IPX providers into the final message. The updated message contents are rewritten into a new HTTP message and sent towards the receiving NF in the SEPP's own PLMN.

```
let N32fRecvSeppA(sepp_a_addr: bitstring)=
    in(n32f_a, (=sepp_a_addr, prins_messsage: modprins));

    (* decompose prins message *)
    let prins''(
        jwe_headers: bitstring,
        nonce: bitstring,
        dataToIntegrityProtect: aad,
        dataToIntegrityProtectAndCipher: bitstring,
        jwe_tag: mac,
        ipx_b_mods: ipxmod,
        ipx_a_mods: ipxmod
    ) = prins_messsage in

    (* decompose aad *)
    let combineAAD(
        source_addr: bitstring,
        destination_addr: bitstring,
        msg_type: bitstring,
        nonconf: bitstring,
        msg_id: nat,
        auth_ipx_id: bitstring,
        n32f_context': bitstring
    ) = dataToIntegrityProtect in

    get storeSeppA(
        sepp_b_plmn, sepp_b_addr, n32f_context, msg_cnt_a, par_req_key_a,
        par_res_key_a, rev_req_key_a, rev_res_key_a, par_req_iv_a,
        par_res_iv_a, rev_req_iv_a, rev_res_iv_a, ciphers_a, encp_a,
        modp_a, modp_b, ipx_a_id, ipx_a_key, ipx_b_id, ipx_b_key
    ) suchthat n32f_context' = n32f_context in

    (* determine session key & nonce *)
    let key = (if msg_type = REQ then
        rev_req_key_a else
        rev_res_key_a) in
    let nonce = (if msg_type = REQ then
        deriveNonce(rev_req_iv_a, msg_id) else
        deriveNonce(rev_res_iv_a, msg_id)) in

    let conf = aeadDecrypt(
        key,
        nonce,
        (dataToIntegrityProtectAndCipher, jwe_tag),
        dataToIntegrityProtect
    ) in
    event recvN32fMsgSeppA(n32f_context', msg_id, conf, nonconf);

    (* verify ipx b json patch *)
    let jsonPatch(
        =modp_b,
        =auth_ipx_id,
        =jwe_tag,
        ipx_b_jws: bitstring
    ) = ipx_b_mods in
    if not(validPrinsSign(ipx_b_jws, ipx_b_key)) then
    (
        event recvInvalIpxPatchSeppA(
            auth_ipx_id, n32f_context, msg_id, jwe_tag, ipx_b_mods
        );
        out(err_a, (n32f_context, msg_id))
    )
    else
        event recvValidIpxPatchSeppA(
            auth_ipx_id, n32f_context, msg_id, jwe_tag, ipx_b_mods
        );

    (* validation of second patch omitted *)

    (* all patches valid *)
    if validPrinsModSign(ipx_a_jws, ipx_a_key) &&
        validPrinsSign(ipx_b_jws, ipx_b_key) &&
        policyValidation(nonconf, encp_a) then
    let msg_body = applyPatches((conf, nonconf), modp_b, modp_a) in
    let http_message = createHttp(
        source_addr,
        destination_addr,
        msg_type,
        msg_body) in
    event sendHttpMsgSeppA(msg_type, msg_body);
    out(plmn_a, (RECV, http_message)).
```

Listing 3.15: Definition of the receiving N32-f signaling transmission process

3.1.4 IPX Processes

As with the SEPPs, the two directly connected intermediaries providers are modeled separately as IPX A and IPX B. Listing 3.16 shows the process of IPX A receiving a message from its peer SEPP, applying the first JSON patch and then sending it on to the next IPX provider.

After receiving the N32-f message, it is decomposed into its individual parts. This operation, among other information, yields the JWE tag required for writing legitimate message modifications. Lines 1114-1116 show how the message operations (i.e. the modifications itself), the IPX provider ID, the JWE tag, and a JWS signature of the complete message are combined to form this patch.

The following decomposition of the AAD object solely serves the purpose of accessing the N32-f context ID and the message ID used by event ipxSendA at the end of the process. Section 4.1 describes how this information is used for verification. Afterwards, the IPX provider's patch is combined with the original message contents in lines 1130-1138 to form a modified PRINS message of type modprins. Note, that there is an additional, empty JSON patch written by this first IPX provider. This is done to simplify the model by not creating a separate data-type for a modified PRINS message with a single JSON patch. Finally, the modified message is sent towards the next IPX provider.

```
let N32fSendIpxA(id: bitstring, privkey_ipx_a: privkey, ops: bitstring)=
    in(n32f_a, (sepp_b_addr: bitstring, prins_message: prins));

    (* decompose prins message *)
    let prins'(
        jwe_headers: bitstring,
        iv: bitstring,
        dataToIntegrityProtect: aad,
        dataToIntegrityProtectAndCipher: bitstring,
        jwe_tag: mac
    ) = prins_message in

    (* compose ipx patch *)
    let modifications = jsonPatch(ops, id, jwe_tag,
        signPrins(prins_message, ops, id, jwe_tag, privkey_ipx_a)
    ) in

    (* decompose aad *)
    let combineAAD(
        source_addr: bitstring,
        destination_addr: bitstring,
        msg_type: bitstring,
        nonconf: bitstring,
        msg_id: nat,
        =id,
        n32f_context: bitstring
    ) = dataToIntegrityProtect in

    (* compose modified message *)
    let modified_message = prins''(
        jwe_headers,
        iv,
        dataToIntegrityProtect,
        dataToIntegrityProtectAndCipher,
        jwe_tag,
        modifications,
        EMPTY
    ) in
    event ipxSendA(n32f_context, msg_id, jwe_tag, modifications);
    out(n32f_i, (sepp_b_addr, modified_message)).
```

Listing 3.16: Definition of the sending IPX process

Listing 3.17 shows the process of IPX A receiving a N32-f message from the IPX network that is to be forwarded to its directly connected SEPP. The actions performed are largely the same as in the sending process above, the difference being that this time, an already modified PRINS message is received on the interface between IPX providers. Further, the empty JSON patch inserted by the previous IPX provider is replaced before forwarding the updated message to the receiving SEPP.

```
let N32fRecvIpxA(id: bitstring, privkey_ipx_a: privkey, ops: bitstring)=
    in(n32f_i, (sepp_a_addr: bitstring, prins_message: modprins));

    (* decompose prins message *)
    let prins''(
        jwe_headers: bitstring,
        iv: bitstring,
        dataToIntegrityProtect: aad,
        dataToIntegrityProtectAndCipher: bitstring,
        jwe_tag: mac,
        ipx_b_mods: ipxmod,
        =EMPTY
    ) = prins_message in

    (* compose ipx patch *)
    let modifications = jsonPatch(ops, id, jwe_tag,
        signPrinsMod(prins_message, ops, id, jwe_tag, privkey_ipx_a)
    ) in

    (* decompose aad *)
    let combineAAD(
        source_addr: bitstring,
        destination_addr: bitstring,
        msg_type: bitstring,
        nonconf: bitstring,
        msg_id: nat,
        auth_ipx_id : bitstring,
        n32f_context: bitstring
    ) = dataToIntegrityProtect in

    (* compose modified message *)
    let modified_message = prins''(
        jwe_headers,
        iv,
        dataToIntegrityProtect,
        dataToIntegrityProtectAndCipher,
        jwe_tag,
        ipx_b_mods,
        modifications
    ) in
    event ipxRecvA(n32f_context, msg_id, jwe_tag, modifications);
    out(n32f_a, (sepp_a_addr, modified_message)).
```

Listing 3.17: Definition of the receiving IPX process

3.1.5 Main Process

Listing 3.18 shows how the main process of the ProVerif model is defined. This is where the previously described processes are combined in a way that models the overall protocol flow. For brevity, the definition of variables, such as idientifiers for SEPP and

IPX nodes, cryptographic keys, and protection policies, have been omitted.

The execution is structured into six separate phases using the keyword sync. As noted in section 2.2, ProVerif synchronization points ensure that processes are executed in a particular order – according to the integer specified behind the sync keyword. Beyond that, all individual processes are combined using the pipe operator „ | ", denoting parallel execution. Hence, processes in one synchronization group are run together and have to terminate successfully before the process continues with the next synchronization group. Processes are not forcefully terminated preemptively.

Processes that are not specifically marked with sync run in the default synchronization group 0. In the created PRINS model this group comprises all N32-c processes necessary to establish a connection between the two SEPPs. Information passed to these processes include PLMN IDs, SEPP addresses as well as the session parameters ciphersuites, protection policies and IPX provider information. All of these information are expected to be preconfigured and remain unchanged in communication with one peer SEPP over one particular IPX provider. In contrast to the rest of the model, the subprocesses comprising the initial N32-c handshake is executed only once by each communicating party. This is to ensure each possible role of a SEPP –i.e. initiating and responding to a N32-c session establishment– is covered, while simultaneously reducing complexity for the rest of the verification by ensuring there is at most two active sessions stored in each of the SEPPs.

Synchronization group 1 (lines 1536-1539) contains the set of processes relevant for creating a new HTTP Request in each of the PLMNs any sending it via N32-f. That is, processes of sending NFs and sending SEPPs on both sides of the inter-operator communication. The NF processes NfARequest and NfBRequest each consume the source and destination address of the message to be created. While in real world deloyments, operators would likely choose to not reveal the IP addresses of their Network Functions and instead provide their partner networks with Fully Quantified Domain Names (FQDNs), this does not make a difference for the purposes of the PRINS verification. The SEPP processes N32fSendSeppA and N32fSendSeppB do not consume any parameters, instead retrieving all necessary information from the previously established N32-f process. Starting from this synchronization group, each process is denoted with a preceeding exclamation mark operator „ ! ". This instructs ProVerif to spawn multiple instances of a given process. By doing so, it is ensured more than one PRINS message can be transferred between the two networks. Furthermore, the tool is able to check for potential security flaws due to confusion of N32 sessions by any of the legitimate participants.

Synchronization group 2 (lines 1541-1549) comprises all IPX processes to transport the previously created N32-f messages. This includes sending and receiving processes by both well-behaving IPX providers A and B as well as IPX providers R1 and R2 intentionally violating the protocol. Chapter 4 further explains how these processes are used to explicitly check some security properties of PRINS. Each of these processes consumes an IPX provider ID and a private key, required to create and sign JSON patches for message modifications as well as the modifications itself. The model assumes that information about the modifications to be performed has been exchanged between the IPX provider and its directly connected SEPP out of band before session establishment and thus, is provided directly to the IPX processes.

Synchronization group 3 (lines 1551-1560) captures the receiving processes of SEPPs and NFs, sending processes of both SEPPs in order to transmit responses to the incoming requests, as well as N32-c error handling processes. The receiving SEPP processes N32fRecvSeppA and N32fRecvSeppB consume their own address in order to identify incoming N32-f messages directed at them from other messages sent by the same SEPP. The same applies to the error handling processes N32cSendErrorSeppA and N32cSendErrorSeppB as well as N32cRecvErrorSeppA and N32cRecvErrorSeppB. The NF processes NfAResponse and NfBResponse, just as the request processes, consume source and destination addresses of the message to be created in response.

Synchronization group 4 and 5 mirror group 2 and 3 in that the response messages created during the previous phase are transmitted via the IPX providers and received by the peer elements. The only exception in synchronization group 5 is that no response is being sent following the receipt of a response message. Hence, sending SEPP processes are omitted.

```
1521      (* process execution *)
1522      (
1523          (N32cSendHandshakeSeppA(
1524              sepp_a_plmn, sepp_a_addr, sepp_b_plmn, sepp_b_addr,
1525              ciphersuites, encp, modp_a, ipx_a_id, ipx_a_pubkey)) |
1526          (N32cRecvHandshakeSeppB(
1527              sepp_b_plmn, sepp_b_addr, sepp_a_plmn, sepp_a_addr,
1528              ciphersuites, encp, modp_b, ipx_b_id, ipx_b_pubkey)) |
1529          (N32cSendHandshakeSeppB(
1530              sepp_b_plmn, sepp_b_addr, sepp_a_plmn, sepp_a_addr,
1531              ciphersuites, encp, modp_b, ipx_b_id, ipx_b_pubkey)) |
1532          (N32cRecvHandshakeSeppA(
1533              sepp_a_plmn, sepp_a_addr, sepp_b_plmn, sepp_b_addr,
1534              ciphersuites, encp, modp_a, ipx_a_id, ipx_a_pubkey)) |
1535
1536          (sync 1; !NfARequest(nf_a_addr, nf_b_addr)) |
1537          (sync 1; !NfBRequest(nf_b_addr, nf_a_addr)) |
1538          (sync 1; !N32fSendSeppA()) |
1539          (sync 1; !N32fSendSeppB()) |
1540
1541          (sync 2; !N32fRecvIpxA(ipx_a_id, ipx_a_privkey, modp_a)) |
1542          (sync 2; !N32fSendIpxA(ipx_a_id, ipx_a_privkey, modp_a)) |
1543          (sync 2; !N32fRecvIpxB(ipx_b_id, ipx_b_privkey, modp_b)) |
1544          (sync 2; !N32fSendIpxB(ipx_b_id, ipx_b_privkey, modp_b)) |
1545          (sync 2; !N32fRecvIpxR1(ipx_a_id, ipx_a_privkey, modp_a)) |
1546          (sync 2; !N32fSendIpxR1(ipx_a_id, ipx_a_privkey, modp_r1)) |
1547          (sync 2; !N32fRecvIpxR2(ipx_r2_id, ipx_r2_privkey, modp_a)) |
1548          (sync 2; !N32fSendIpxR2(ipx_r2_id, ipx_r2_privkey, modp_a)) |
1549          (sync 2; !N32fRecvIpxR3()) |
1550
1551          (sync 3; !N32fRecvSeppA(sepp_a_addr)) |
1552          (sync 3; !N32fRecvSeppB(sepp_b_addr)) |
1553          (sync 3; !NfAResponse(nf_a_addr, nf_b_addr)) |
1554          (sync 3; !NfBResponse(nf_b_addr, nf_a_addr)) |
1555          (sync 3; !N32cSendErrorSeppA(sepp_a_addr)) |
1556          (sync 3; !N32cRecvErrorSeppA(sepp_a_addr)) |
        (sync 3; !N32cSendErrorSeppB(sepp_b_addr)) |
        (sync 3; !N32cRecvErrorSeppB(sepp_b_addr)) |
        (sync 3; !N32fSendSeppA()) |
        (sync 3; !N32fSendSeppB()) |

        (sync 4; !N32fRecvIpxA(ipx_a_id, ipx_a_privkey, modp_a)) |
        (sync 4; !N32fSendIpxA(ipx_a_id, ipx_a_privkey, modp_a)) |
        (sync 4; !N32fRecvIpxB(ipx_b_id, ipx_b_privkey, modp_b)) |
        (sync 4; !N32fSendIpxB(ipx_b_id, ipx_b_privkey, modp_b)) |
        (sync 4; !N32fRecvIpxR1(ipx_a_id, ipx_a_privkey, modp_a)) |
        (sync 4; !N32fSendIpxR1(ipx_a_id, ipx_a_privkey, modp_r1)) |
        (sync 4; !N32fRecvIpxR2(ipx_r2_id, ipx_r2_privkey, modp_a)) |
        (sync 4; !N32fSendIpxR2(ipx_r2_id, ipx_r2_privkey, modp_a)) |
        (sync 4; !N32fRecvIpxR3()) |

        (sync 5; !N32fRecvSeppA(sepp_a_addr)) |
        (sync 5; !N32fRecvSeppB(sepp_b_addr)) |
        (sync 5; !NfAResponse(nf_a_addr, nf_b_addr)) |
        (sync 5; !NfBResponse(nf_b_addr, nf_a_addr)) |
        (sync 5; !N32cSendErrorSeppA(sepp_a_addr)) |
        (sync 5; !N32cRecvErrorSeppA(sepp_a_addr)) |
        (sync 5; !N32cSendErrorSeppB(sepp_b_addr)) |
        (sync 5; !N32cRecvErrorSeppB(sepp_b_addr))
    )
```

Listing 3.18: Definition of the main process

3.2 Deviations from the Specification

3.2.1 Deliberate Abstractions

The ProVerif model of PRINS described in the previous section necessarily abstracts away a number of details that have to be considered by real-world implementations. In the following, these intentional abstractions and their implications on the verification of security properties are discussed.

Perfectly Secure TLS & Key Derivation

As outlined before, this analysis solely focusses on the novel parts of the PRINS protocol defined by 3GPP. TLS is assumed to be perfectly secure and thus, is not modeled in detail. Instead, TLS-based connections are represented by secure (i.e. confidential and integrity protected) channels shown in listing 3.2. Since mutual authentication in N32-c communication crucially depends on the security of the transport layer, this model is not able to effectively validate this property of N32-c.

Another protocol aspect closely coupled with the TLS protocol is derivation of cryptographic key material during the N32-c handshake procedure. According to the specification „two SEPPs shall export keying material from the TLS session established between them using the TLS export function" (3GPP 2019c, p. 132). This export function produces a pseudorandom bitstring that is used to derive the various N32-f session keys outlined in section 2.1. The model's N32-c channel being inherently secure by assumption, it is not possible to model a similar function producing fresh key material based on a shared, authenticated session. This issue is being addressed, by the private constructor deriveMasterKey, consuming the N32-f context ID. This input parameter ensures that the resulting master key is unique to a specific N32-f session. The attribute private guarantees that this function is not be accessible to adversaries, so that even with possession of the N32-f context ID, it is not possible to derive the same cryptographic key. This kind of substitution for the TLS export function is flawed in two ways. Firstly, it relies on the secrecy of the derivation function iteslf, rather than the secrecy of the input parameter. Secondly, the attribute private still allows the function to be used by all legitimate protocol participants. In the created model, this also includes the IPX processes. However, these points do not impact verification of the protocol, as private ProVerif functions cannot be leaked and the honest IPX processes do not make use of the function either.

Protection Policies

PRINS protection policies control what data-types are to be ciphered before being sent over the N32-f channel and which authenticated modifications may be performed on them. Both protection policies are abstracted to single bitstrings to simplify related operations.

The data-type encryption policy plays a key role in rewriting HTTP messages received from NFs in the SEPP's own PLMN. In the model, confidential and non-confidential message parts are extracted from an HTTP message body using the constructors getConf and getNonconf, displayed in listing 3.19. Using an encryption policy as a second argument, they enable honest participants to identify those information elements only requiring integrity protection and those additionally requiring confidentialiy. Note, that since the model does not consider protection policies confidential information and makes them publicly available, a flawed representation is avoided in which the adversary would not be able to retrieve sensitive information from the body of an HTTP message without the related encryption policy.

Before the receiving SEPP compiles the resulting HTTP message, it uses the policyValidation deconstructor to verify whether the sending peer actually applied the previously agreed encryption policy or whether sensitive elements are actually contained in non-confidential parts of the N32-f message. This is in addition to the equality check of both locally stored and received encryption policies during the initial N32-c handshake.

```
fun getConf(bitstring, bitstring): bitstring.
fun getNonconf(bitstring, bitstring): bitstring.

fun policyValidation(bitstring, bitstring): bool
reduc forall m: bitstring, p: bitstring;
    policyValidation(getNonconf(m, p), p) = true
otherwise forall m: bitstring, p': bitstring, p: bitstring;
    policyValidation(getNonconf(m, p'), p) = false
otherwise forall m: bitstring, p': bitstring, p: bitstring;
    policyValidation(fail, p) = false.
```

Listing 3.19: N32-f Message Rewriting Functions

The bitstrings modeling the modification policies represent both the policies defined by either of the SEPP operators as well as the actual modifications written by intermediary IPX providers. This allows a simple comparison of the operation contained inside the JSON patch and the related policy itself to determine whether or not a given message modification is legitimate. This does not impact the verification result, as it is logically the same as a policy containing exactly one entry. If the protocol behaves as intended using this examplary modification policy, it behaves similarly for more complex examples.

Message Routing

As mentioned in the previous section, inter-PLMN message routing, which would utilize IP routing in real-world implementations, is simplified greatly by use of dedicated N32-c and N32-f channels. Processes of sending SEPPs specify their communication peer by its address in outgouing messages. This information is provided to avoid trivial loops between sending and receiving processes of the same SEPP: Without any attribute identifying either part of the communication, an outgoing message could be consumed by an instance of the same SEPP's receiving process without noticing the message is in fact its own. Hence, in the model receiving both SEPPs verify whether the identity specified in incoming messages matches their own and only proceed with processing the message if this condition is met. Note that this does not preclude the verification of potential authentication flaws – an adversary is, of course, still able to spoof the SEPP address value. The check merely prevents the confusion of messages by honest SEPP processes.

Beyond that, the model does not consider any information identifying the source of a message. That is, at least on the N32-c channel, which is considered mutually authenticated as explained previously. As far as N32-f communication is concerned, the receiving SEPP is able to correlate an incoming message by its N32-f context ID to the locally stored session parameters. By validating the AAD contained in the message, the SEPP is able to implicitly verify the authenticity of the original message contents.

IPX Provider Internals

Similarly to message routing from and to communicating SEPPs, the message handling by IPX providers is reduced to the bare minimum. Each IPX process utilizes two distinct N32-f channels, one towards its directly connected SEPP and another towards the second IPX provider. Every message received on one of these interfaces is forwarded on the other one after internal processing.

Modifications are signed using raw private keys, one of two options specified by the 3GPP specification for this purpose. The alternative, i.e. a digital certificate issued by the operator of the directly connected SEPP, as well as the possibility for IPX providers to use multiple cryptographic identities is not considered in the model. Since there is no fundamental difference between the use of one or more keys and the asymmetric cryptography used in digital certificates is the same as for raw private/public key pairs, this abstraction does not impact the verification result.

Parameter Renegotiation

Renegotiation of session parameters is left out of scope of this work entirely. Such procedure is required when existing session keys reach the end of their lifetime or any other preconfigured session parameter, such as the cipher suite to be used, changes. Firstly, at the time of this writing the 3GPP specifications TS 33.501 and TS 29.573 do not clearly stipulate how a SEPP is to perform a renegotiation. Secondly, the triggers that would necessitate this procedure, such as the timeout of cryptographic keys are not considered in the model either, for reasons described in the following section.

3.2.2 Limitations of ProVerif

Invalidation of N32-f Keys & Contexts

Formal verification in general provides very limited support for expressing temporal relations. While ProVerif allows for the verification of correspondence assertions, ensuring that certain events do or do not happen after other events, it does not allow to convey precise timings. Hence, the model does not consider the limited lifetime of cryptographic keys associated with a N32-f context.

Furthermore, real-world protocol implementations relying on incremental counters, such as the message IDs used in N32-f communication, are forced to handle limits of integer values to avoid overflows. At some point during protocol execution, session parameters have to be refreshed in order to prevent counter wrap around. ProVerif does offer support for natural numbers, however, the concept of integer overflows is not reproduceble.

Lastly, ProVerif tables, which represent a convenient way of modeling local SEPP storage as shown in the previous sections, do not offer the possibility of deleting records. While real-world implementations would invalidate N32-f contexts at certain events or specific points in time, the model is not able to reproduce this behavior.

Behavior of N32-c Channels

According to the specification, the N32-c is unidirectional with regards to the initiation of a message exchange. While a peer may respond on the same logical channel, only the initiating SEPP can actively issue requests. This is due to the nature of the underlying HTTP protocol, which follows a client-server model. The initiating SEPP takes on the role of the client, the responding SEPP that of the server. In order to enable the responding end to initiate requests as well, for example to perform error signaling, a second N32-c connection in the inverse direction is established at the end of the initial handshake.

ProVerif channels, no matter if private or not, are always bidirectional. While this fails to restrict messages being sent in either direction, one can work around this limitation during the modeling phase by letting sending and responding processes of both SEPPs use separate N32-c channels. Since private channels, unlike public ones, cannot be created ad hoc during the protocol execution, both N32-c channels have to be declared in advance.

An additional characteristic of private channels that does not mirror a real-world TLS connection is that every information sent over them can be received by any of the listening participants at any time. As the user manual explains, „ProVerif just computes a set of messages sent on a private channel, and considers that any input on that private channel can receive any of these messages (independently of the order in which they are sent)” (Blanchet, Smyth, et al. 2020, p. 119).

Chapter 4

Verifying the PRINS Protocol

The following chapter details the verification performed on the PRINS model described in the preceeding chapter. Section 4.1 describes the formalization of security properties outlined in 2.1.4 and the results ProVerif produces. Section 4.2 describes challenges faced during the verification process and lessons learned. Section 4.3 summarizes gaps and inconsistencies discovered during modelling and verification of PRINS and what they mean for the security of the protocol.

4.1 ProVerif Queries

4.1.1 Reachability Queries

Formal verification of ProVerif models evolves around events. Events are defined points in the protocol execution that mark certain states of interest. During verification, the software verifies whether these can be successfully reached and in what number and order they correspond to each other.

Reachability queries check whether a certain state can actually be obtained during a protocol run. Events can also be combined using the logical operators AND (&&) and OR (||) to test whether a certain combination is reachable or not. Due to this nature, they are used in conjuction with the created model mainly to prove the absence of undesired events or a combination of certain events that is undesired.

Debug Queries

A common mistake in formal verification is modeling the system in such a way that parts or even the whole protocol are not executed at all, leading to erroneous results that convey a false sense of security. For this reason, simple reachability queries are defined for all defined events in the protocol, merely verifying whether each event is individually reachable. If this is not the case, this indicates that the protocol execution stops prior to the specified point and never reaches the desired state. These debug queries are defined in the format `query event(e)`, where `e` is the respective event under test. As they are only used for the purposes of validating the correctness of the created model and all follow the same pattern, they are not depicted hereafter. Important to highlight is that all events in the model are indeed reachable, meaning there are no protocol branches which can never be executed.

Secrecy Queries

ProVerif secrecy queries are essentially reachability queries on terms rather than events. They verify whether the Dolev-Yao adversary is able to successfully compute a certain value using public information. As detailed in the user manual document, ProVerif offers two ways of specifying such queries – using the keyword attacker or using the keyword secret. The former checks „whether the attacker can compute the term M, built from free names" (Blanchet, Smyth, et al. 2020, p. 55), while the latter is used to express queries that test the secrecy of the variable or bound name specified. Since the created model uses special names that hold all confidential information during protocol execution, only the second variant is used.

There are four types of data in the PRINS protocol which are crucially confidential: private keys of both SEPPs and IPX nodes, master session keys exported from the N32-f TLS session, session keys derived from them, and the confidential contents in N32-f messages. As described in section 3.2, TLS connections are assumed to be secure and thus, there is no need for the protocol to consider private and public keys of the communicating SEPPs.

Listing 4.1 below shows all secrecy queries defined for the PRINS protocol. Initially, the above-mentioned information elements are defined as global variables assumed to be private, i.e. not available to the adversary. It should be noted that although this property is assumed to be true, ProVerif does verify the secrecy of these names and fails with an error message should it be able to disprove the assumption (Blanchet, Smyth, et al. 2020, p. 111).

```
free privkey_ipx_a, privkey_ipx_b: privkey [private].
free par_req_key_a, par_res_key_a, rev_req_key_a, rev_res_key_a,
    par_req_key_b, par_res_key_b, rev_req_key_b, rev_res_key_b,
    master_key_a, master_key_b: key [private].
free conf: bitstring [private].

(* ipx key confidentiality *)
query secret privkey_ipx_a [reachability];
    secret privkey_ipx_b [reachability].

(* master & session key confidentiality *)
query secret master_key_a [reachability];
    secret master_key_b [reachability];
    secret par_req_key_a [reachability];
    secret par_res_key_a [reachability];
    secret rev_req_key_a [reachability];
    secret rev_res_key_a [reachability];
    secret par_req_key_b [reachability];
    secret par_res_key_b [reachability];
    secret rev_req_key_b [reachability];
    secret rev_res_key_b [reachability].

```

```
257 (* sec property 7: message confidentiality *)
258 query secret conf [reachability].
```

Listing 4.1: Secrecy Queries

As they are the cryptographic basis for authenticating message modifications, the secrecy of IPX providers' private keys is a crucial prerequisite for ensuring only legitimate intermediaries may perform changes. Hence, proving unreachability of privkey_ipx_a and privkey_ipx_b to the adversary in the created model is the goal of the first two secrecy queries in lines 242-243. As these variables are only ever consumed by the IPX providers themselves and do not get sent across any channel whatsoever, this is a fairly trivial assertion and indeed, ProVerif is able to successfully prove unreachability to the attacker.

The next secrecy queries concern the master keys and session keys used by both SEPPs for the N32-f AEAD encryption operations. While the compromise of a session key would only directly result in a lack of confidentiality for part of the communication –due to cryptographic separation based on the sending or receiving role of the SEPP as well as the transported message being a request or response–, a master key compromise would allow the adversary to derive the complete hirarchy of cryptographic keys and IVs. As described previously, master keys are created using the private constructor deriveMasterKey. Afterwards, they are only used for further derivation of cryptographic material and never sent on a channel. In contrast, session keys derived from these master keys are stored in the private tables of each of the SEPPs and used subsequently by the aeadEncrypt and aeadDecrypt destructors. Based on the created model, ProVerif is able to successfully prove that adversaries are unable to compute any of this cryptographic material, thereby validating an important prerequisite for N32-f message confidentiality.

Lastly, above query query secret conf [reachability] explicitly validates confidentiality of the special variable conf which stores sensitive contents of signaling messages as per the encryption policy. This corresponds to security property 7 in section 2.1 – the confidentiality of information contained in the dataToIntegrityProtectAndCipher block of N32-f messages. This property, too, is successfully proven by ProVerif, confirming that the AEAD encryption and decryption operations are modeled in a way that preserves confidentiality between SEPPs.

In conclusion, the above queries are able to prove on the basis of the created model that all information requiring secrecy in the PRINS protocol is indeed never acccessible to the adversary.

Dishonest participants

In order to explicitly test some of the security properties listed in section 2.1.4, several protocol agents are modeled that intentionally violate the intended protocol behavior. These misbehaving agents comprise sending and receiving processes of the rogue IPX providers R1, R2, and R3.

Security property 8 states that intermediaries shall not be able to remove previously added message modifications. In order to prove this requirement, the IPX process N32fRecvIpxR1 is introduced, which violates the protocol by doing exactly that. Similar to the process N32fRecvIpxA shown in listing 3.17, the only difference between the two is that IPX R1 replaces the existing patch object with the empty patch. The query in listing 4.2 below asserts that the events recvValidIpxPatchSeppA and ipxRecvR1 –given the same N32-f context ID, message ID, JWE tag, and JSON patch– are not reachable together. That is, all messages from by IPX R1 never result in a patch being recognized as valid by the receiving SEPP.

ProVerif is able to disprove this query, returning a negative result. This means both states can indeed by reached together in the same execution of the program. Trying to export a representation of the discovered trace, however, results in non-termination of the program. This is not due to a limitation of ProVerif itself but of the *Graphviz* software, specifically its *dot* utility used for drawing directed graphs of the attack trace. Due to the large number of protocol states, the program is unable to finalize the trace even after several hours of execution.

At this point, it should be noted that not all attacks ProVerif discovers are indeed applicable or practical in real-world scenarios. Since the attack trace cannot be produced, there is a chance the program finds a wrong attack. One possible scenario that may be discovered as an attack, since the query does not take into account any temporal relation between the two events, is the SEPP receiving a valid message first, reaching the event recvValidIpxPatchSeppA and the IPX R1 process receiving the same message by the adversary who is replaying the message afterwards. However, without further details about the trace found by ProVerif, it is not possible to validate this assumption.

```
(* sec property 8: removing previous message modifications *)
query ipx_b_id: bitstring, n32f_context: bitstring,
    msg_id: nat, jwe_tag: mac, ipx_a_mods: ipxmod, ipx_b_mods: ipxmod;

    event(recvValidIpxPatchSeppA(
        ipx_b_id, n32f_context, msg_id, jwe_tag, ipx_b_mods)) &&
    event(ipxRecvR1(n32f_context, msg_id, jwe_tag, ipx_a_mods)).
```

Listing 4.2: Query for security property 8

Listing 4.3 involves the intentionally misbehaving IPX provider R3 to verify security property 10. According to the 3GPP specification, an adversary on the inter-operator network shall not be able to record message modifications by legitimate IPX providers and re-apply them to subsequent messages at a later point in time. Since intermediary message modifications must contain the JWE tag of the original message before being signed, they are only valid for that particular message. The process N32fRecvIpxR3, too, takes a similar role as N32fRecvIpxA. Connected to both IPX B via the channel n32f_i and to SEPP A via the channel n32f_a, it tries to re-apply the already signed modifications by IPX B before the event ipxRecvR3 is triggered and the message is sent towards SEPP A.

ProVerif proves this query wrong, returning a negative result. Since the same limitation for the export of the discovered attack trace described above applies as well, it is unclear whether this corresponds to a real attack. One potential source of a false attack trace is again the lack of a specified temporal relation, which allows for the possibility that ipxRecvR3 is executed after recvValidIpxPatchSeppA.

```
(* sec property 10: rejecting replayed json patches *)
query ipx_a_id: bitstring, n32f_context: bitstring,
    msg_id: nat, jwe_tag: mac, ipx_mods: ipxmod;

    event(recvValidIpxPatchSeppA(
        ipx_a_id, n32f_context, msg_id, jwe_tag, ipx_mods)) &&
    event(ipxRecvR3(n32f_context, msg_id, jwe_tag, ipx_mods)).
```

Listing 4.3: Query for security property 10

Listing 4.4 shows the query for security property 11, i.e. the receiving SEPP only accepting message modifications from known IPX providers. The process of IPX R2 behaves similarly to the sending and receiving processes of the genuine IPX A in that it consumes N32-f messages, applies exactly the same message modifications, and forwards the message towards either of the SEPPs. The only difference is the IPX ID and the private key used by IPX R2 are not known to the receiving party.

ProVerif successfully proves this property true, returning a positive result. This means no trace of the protocol model has been found in which SEPP A or SEPP B accept modifications written by IPX R2.

```
(* sec property 11: rejecting unknown ipx providers *)
query ipx_a_id: bitstring, n32f_context: bitstring,
    msg_id: nat, jwe_tag: mac, ipx_a_mods: ipxmod;
    event(recvValidIpxPatchSeppA(
        ipx_a_id, n32f_context, msg_id, jwe_tag, ipx_a_mods)) &&
    event(ipxRecvR2(n32f_context, msg_id, jwe_tag, ipx_a_mods));

    event(recvValidIpxPatchSeppB(
        ipx_a_id, n32f_context, msg_id, jwe_tag, ipx_a_mods)) &&
    event(ipxSendR2(n32f_context, msg_id, jwe_tag, ipx_a_mods)).
```

Listing 4.4: Query for security property 11

The query in listing 4.5 tests security property 12, i.e. the receiving SEPP rejecting JSON patches that contain message modifications not captured in the related modification policy. If the patch originates from the foreign SEPP's IPX provider, then the foreign SEPP's modification policy applies. If the patch is written by the directly connected IPX provider, the SEPP's own modification policy is used. The sending process of IPX R1 behaves similarly as the the genuine sending process of IPX A, including the use of the same IPX ID and private key for signing message modifications. However, the modifications written as part of the JSON patch do not match the modification policy the receiving SEPP uses for verification.

ProVerif is able to successfully validate this property, returning a positive result. This proves the absence of a trace in which IPX R1 writes message modifications and SEPP B recognizes them as being valid.

```
(* sec property 12: rejecting unknown message modifications *)
query ipx_a_id: bitstring, n32f_context: bitstring,
    msg_id: nat, jwe_tag: mac, ipx_mods: ipxmod;

    event(recvValidIpxPatchSeppB(
        ipx_a_id, n32f_context, msg_id, jwe_tag, ipx_mods)) &&
    event(ipxSendR1(n32f_context, msg_id, jwe_tag, ipx_mods)).
```

Listing 4.5: Query for security property 12

4.1.2 Correspondence Assertions

Correspondence assertions verify in which order and multiplicity events related to each other. As outlined previously, they can be used to test integrity and authentication properties such as security properties 1 and 2 in section 2.1.4. However, given the assumptions made in modeling the PRINS protocol, these particular requirements are not explicitly addressed. Since the three main security properties of N32-c communication –mutual authentication, integrity and confidentiality– are all provided by TLS, there is no way to meaningfully prove them in the created model.

One of the properties successfully proven using correspondence assertions is security property 3, i.e. the derivation of the same N32-f context ID during the initial N32-c handshake. Listing 4.6 below shows the query for the same, taking into account both the sending and receiving role for each of the SEPPs. Following the exchange of their respective pre-context IDs over N32-c, both peers combine the information using the deriveContextID constructor with the sending SEPP's input first and the receiving SEPP's input second. Afterwards, the events sendN32fContext and recvN32fContext are triggered, depending on the SEPP's role. The related queries assert that both SEPPs only reach these states when the resulting N32-f context IDs are the same. As the absence of a trace that does not satisfy this condition is successfully proven, it is guarateed that both SEPP always derive a matching N32-f context ID.

```
(* sec property 3: N32-f Context IDs equivalent *)
query sepp_a_addr: bitstring, sepp_b_addr: bitstring,
    n32f_a_pid: bitstring, n32f_b_pid: bitstring,
    n32f_context_a: bitstring, n32f_context_b: bitstring;
    event(recvN32fContext(
        sepp_b_addr, n32f_a_pid, n32f_b_pid, n32f_context_b
    )) &&
    event(sendN32fContext(
        sepp_a_addr, n32f_a_pid, n32f_b_pid, n32f_context_a
    )) ==>
    n32f_context_a = n32f_context_b;

    event(recvN32fContext(
        sepp_a_addr, n32f_b_pid, n32f_a_pid, n32f_context_a
    )) &&
    event(sendN32fContext(
        sepp_b_addr, n32f_b_pid, n32f_a_pid, n32f_context_b
    )) ==>
    n32f_context_a = n32f_context_b.
```

Listing 4.6: Query for security property 3

The next security property 4 concerns the derivation of master keys. Both sending and receiving SEPP shall derive matching keys during the initial N32-c handshake. ProVerif is able to confirm this requirement using the queries shown in listing 4.7 below. As described in section 3.2.1, the model represents the TLS export function by a private constructor. Due to this over-abstraction, the insights gained into the PRINS protocol itself are rather limited. Nevertheless, proving the fact that sending and receiving SEPP only reach the protocol states sendMasterKey and recvMasterKey if the resulting master keys are equal does increase the confidence in the correct behavior of the created model.

```
(* sec property 4: master keys equivalent *)
query sepp_a_addr: bitstring, sepp_b_addr: bitstring, n32f_cid: bitstring,
    mkey_a: key, mkey_b: key;
    event(recvMasterKey(sepp_b_addr, n32f_cid, mkey_a)) &&
    event(sendMasterKey(sepp_a_addr, n32f_cid, mkey_b)) ==>
    mkey_a = mkey_b;

    event(recvMasterKey(sepp_a_addr, n32f_cid, mkey_a)) &&
    event(sendMasterKey(sepp_b_addr, n32f_cid, mkey_b)) ==>
    mkey_b = mkey_a.
```

Listing 4.7: Query for security property 4

Listing 4.8 shows a correspondence query addressing multiple protocol requirements at the same time. Security properties 6 and 7 require all N32-f messages to be authenticated and all message contents to be integrity protected.

The events sendN32fMsgSeppB and recvN32fMsgSeppA as well as sendN32fMsgSeppA and recvN32fMsgSeppB depicted below should always be executed after another, given the same N32-f context ID, message ID, and message contents. Moreover, they should demonstrate injective correnspondence, meaning each receiving event is preceeded by exactly one sending event in the foreign SEPP. Hence, the two sending events are marked with the keyword inj-event. The parameters msg_conf and msg_nonconf represent the *dataToIntegrityProtectAndCipher* and *dataToIntegrityProtect* objects of an N32-f message.

Attempting to prove this property results in non-termination of ProVerif. While the program was still responsive and did make occasional progress, the execution of ProVerif was aborted after 100 hours of runtime. This shows one of the limitations of model checking of unbound protocol sessions, described in more detail in the following section: Ensuring the verification terminates is not trivial. In order to be able to address the remaining queries, this particular test has to be removed from the ProVerif file, leaving the integrity protection and authentication of N32-f messages unverified.

```
(* sec property 6/7: integrity protection / authentication *)
query msg_id: nat, n32f_context: bitstring,
    msg_conf: bitstring, msg_nonconf: aad;
    event(recvN32fMsgSeppA(
        n32f_context, msg_id, msg_conf, msg_nonconf)) ==>
    inj-event(sendN32fMsgSeppB(
        n32f_context, msg_id, msg_conf, msg_nonconf));

    event(recvN32fMsgSeppB(
        n32f_context, msg_id, msg_conf, msg_nonconf)) ==>
    inj-event(sendN32fMsgSeppA(
        n32f_context, msg_id, msg_conf, msg_nonconf)).
```

Listing 4.8: Query for security property 6 and 7

Listing 4.9 shows the query testing security property 9, i.e. authentication and integrity protection of IPX message modifications. It is defined as an injective correspondence between two successful patch validations in the receiving SEPP (recvValidIpxPatchSeppA or recvValidIpxPatchSeppB) and two sending and receiving IPX provider processes (ipxRecvA, ipxSendB, ipxRecvB, and ipxSendA). In other words, given the same N32-f context ID and message ID, each successful validation of both IPX patches must be preceeded by the directly connected IPX provider receiving this message and the peer SEPP's IPX provider sending the same.

For both of these queries ProVerif returns a negative result, meaning that a trace has been found in which the receiving SEPP identifying two valid patches from different IPX providers is not preceeded by IPX A and IPX B transmitting the same message. As for the other discovered attacks, the visual trace graphs again cannot be successfully created.

```
(* sec property 9: json patch authentication/integrity protection *)
query n32f_context: bitstring, ipx_a_id: bitstring, ipx_b_id: bitstring,
    msg_id: nat, jwe_tag: mac, ipx_a_mods: ipxmod, ipx_b_mods: ipxmod,
    http_method: bitstring, msg_body: bitstring;
    event(recvValidIpxPatchSeppA(
        ipx_a_id, n32f_context, msg_id, jwe_tag, ipx_a_mods)) &&
    inj-event(recvValidIpxPatchSeppA(
        ipx_b_id, n32f_context, msg_id, jwe_tag, ipx_b_mods)) ==> (
            inj-event(ipxRecvA(n32f_context, msg_id, jwe_tag, ipx_a_mods))
            &&
            inj-event(ipxSendB(n32f_context, msg_id, jwe_tag, ipx_b_mods))
        );

    event(recvValidIpxPatchSeppB(
        ipx_b_id, n32f_context, msg_id, jwe_tag, ipx_b_mods)) &&
    inj-event(recvValidIpxPatchSeppB(
        ipx_a_id, n32f_context, msg_id, jwe_tag, ipx_a_mods)) ==> (
            inj-event(ipxRecvB(n32f_context, msg_id, jwe_tag, ipx_b_mods))
            &&
            inj-event(ipxSendA(n32f_context, msg_id, jwe_tag, ipx_a_mods))
        ).
```

Listing 4.9: Query for security property 9

4.1.3 Observational Equivalence

The 3GPP specification does not cite specific observational equivalence requirements. Nevertheless, as highlighted by security property 5 in section 2.1, strong secrecy for the master keys and derived session keys is certainly desireable, meaning the attacker should not be able to detect when any of these parameters change. ProVerif generally features verification of strong secrecy by use of the noninterf keyword (Blanchet, Smyth, et al.

2020, p. 55). However, modeling the PRINS protocol exposes multiple corner cases for which verification of this property is presently not supported. These shortcomings lead to a failure to successfully verify strong secrecy for the created model of PRINS with the queries shown in listing 4.10 below.

```
(* sec property 5: strong secrecy of master/session keys *)
noninterf master_key_a.
noninterf master_key_b.
noninterf par_req_key_a.
noninterf par_res_key_a.
noninterf rev_req_key_a.
noninterf rev_res_key_a.
noninterf par_req_key_b.
noninterf par_res_key_b.
noninterf rev_req_key_b.
noninterf rev_res_key_b.
```

Listing 4.10: Query for security property 5

An initial version of the model implements N32-f message IDs as natural numbers. This allows increasing the sequence counter with every outgoing message and convenient checks whether a certain N32-f message ID was already sent previously. When attempting to verify this model with ProVerif, the tool outputs the error message *„Natural numbers do not work with non-interference yet"*. Hence, any natural number in the model would impede strong secrecy queries.

An alternative variant of the model created due to the above limitation represents N32-f message IDs using bitstrings. In this case, incrementing the sequence counter within the SEPP is a matter of calling a trivial constructor consuming a bitstring (the old sequence number) and producing another one (the updated sequence number). The check for previously sent message IDs can be modeled using a set of bitstrings, which is updated every time a N32-f message is sent out. A suitable way to determine set membership using ProVerif's predicate construct is described in the official user manual (Blanchet, Smyth, et al. 2020, p. 96). However, program execution using this representation of message IDs yields the following error message: *„Predicates are currently incompatible with non-interference"*.

Modeling N32-f message IDs as bitstrings does not result in a improvement of the verification result. The final model of the protocol uses natural numbers, since this resembles real-world implementations more closely and further helps to reduce the amount of code required to model this particular protocol aspect, thereby reducing complexity and the potential for errors.

4.1.4 Summary

All in all, the queries described above are only able to successfully verify four out of the 12 security properties listed in section 2.1.4. A formal proof for the remaining two thirds cannot be produced due several different reasons.

As N32-c is modeled using ProVerif private channels without considering how the same are established, proofs for security property 1 and 2 are superfluous. Given that TLS has successfully been proven secure in previous research (see Cremers et al. 2017, Merwe 2018), it is fair to assume that these requirements are met. The strong secrecy property cannot be proven by ProVerif due to a limitation of the tool itself, regardless of the N32-f message ID being modeled as natural number or bitstring. Properties 6 and 7 lead to non-termination of ProVerif itself while properties 8, 9, and 10 are explicitly disproven by the program, but a visual represenation of the attack traces cannot be exported due to the large number of nodes and edges in the resulting graph.

Table 4.1 summarizes the results of the ProVerif queries applied to the created model of PRINS.

Security Property	Query	Result	Comment
1	None	True	True by assumption
2	None	True	True by assumption
3	Listing 4.6	True	Successfully proved
4	Listing 4.7	True	Successfully proved
5	Listing 4.10	Unclear	ProVerif limitation
6	Listing 4.8	Unclear	ProVerif does not terminate
7	Listing 4.8	Unclear	ProVerif does not terminate
8	Listing 4.2	False	Disproved, attack trace not available
9	Listing 4.9	False	Disproved, attack trace not available
10	Listing 4.3	False	Disproved, attack trace not available
11	Listing 4.4	True	Successfully proved
12	Listing 4.5	True	Successfully proved

Table 4.1: Verification result of the PRINS security properties

While it is positive to successfully prove part of the properties, these results offer limited insights into why the verification of the majority of properties fails or is inconclusive. One can argue that an attack reconstruction is actually an essential part of a formal verification as to guide potential improvements to the protocol under test. However, all throughout the verification of the protocol there have been a number of significant challenges in ensuring the model does behave as intended and composing queries that produce a meaningful result. Some of these have been resolved as part of this thesis, others remain open and may be subject for future work. The next section details the user experience of ProVerif for this particular project and lessons learned for future verification attempts.

4.2 ProVerif Verification Experience

Aside from insights into the security properties of PRINS itself, modeling a complex protocol like this also provides a better understanding of the tool ProVerif. The following is a brief account of the key challenges encountered during the creation this thesis.

Firstly, one of the most challenging aspects of the verification with ProVerif is the lack of built-in debug capabilities. The reason why the debug queries in section 4.1.1 are introduced to the verification is that without them, there is no indication when specified events cannot be reached individually. Instead, compound reachability queries or correspondence assertions that incorporate these events are likely to provide erroneous results.

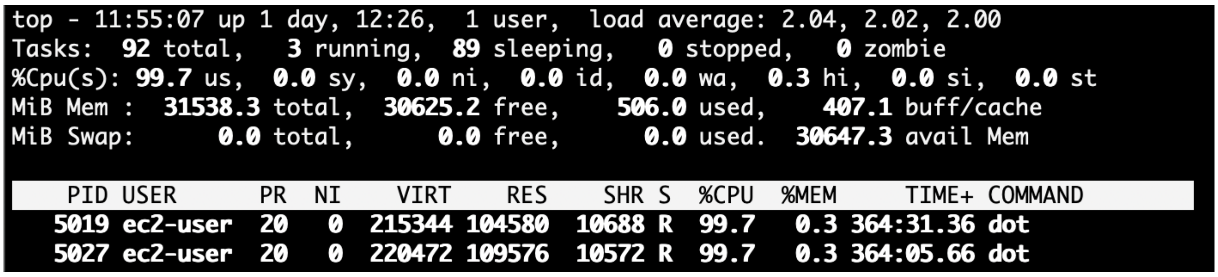
Without being aware of this behavior and explicitly testing the reachability of every single event itself, this may at times lead to a false confidence into the properties of the protocol. Thankfully, the ProVerif user manual does mention using simple reachability queries to prevent exactly that (see Blanchet, Smyth, et al. 2020, p. 52). A lesson learned from facing the issue of certain protocol branches being unreachable is to add said debug queries from the very beginning of the modeling process each time a new event is specified.

Secondly, ensuring termination of the model checker is all but trivial and the tool provides little insights into situations that may lead to non-termination. One commonly encountered issue during the verification of PRINS, such as in the case of the query in listing 4.8, is the program inserting an increasing number of rules during the resolution process when trying to prove a given fact is derivable from available clauses, but never succeeding to do so. In other scenarios the program does not print out any indication of progress anymore and seemingly halts.

One source of this problem during verification is the confusion of sent and received messages between processes, leading to loops within the protocol. As soon as the type signature of a ProVerif function matches those of the avilable names, the tool will attempt to execute it. The recommended way to avoid such mix-ups is tagging the protocol using defined constants to guide the execution of honest protocol participants, as explained in the user manual by example of a cryptographic constructor (see Blanchet, Smyth, et al. 2020, p. 114). Key takeaway from the modeling experience of PRINS is to add tags to every single input and any custom functions that may be ambiguous based on the type of their input arguments.

Beyond general best practices like the one above, ProVerif can offer little support to prevent non-termination, as termination analysis is generally an undecidable problem. The configuration flag verboseTerm which is supposed to enable a more detailed output with termination warnings does not yield any helpful results for the problematic queries. Different built-in resolution strategies do not solve the problem either.

Thirdly, even if ProVerif itself successfully terminates and finds an attack, there is no guarantee the related trace can be exported in form of a visual graph. By default, the program creates files of type dot that can be consumed by the software Graphviz to draw a directed graph into several different output formats, such as pdf, png, or svg. With large protocol models such as the one created for PRINS, Graphviz's DOT utility is quickly overburdened by the number of nodes representing key protocol events and edges representing their relation as well as message flows. As shown in figure 4.1 after more than six hours of execution, Graphviz is still not able to completely render an attack trace from any of the disproven queries described in the previous sections. Omitting all replications in the main process of the model, denoted by an exclamation mark, does not result in a noticable improvement. Since DOT is the only utility in the Graphviz toolbox that produces directed graphs, the output of other programs such as SFDP or NEATO –which do terminate– is not suitable to retrace the protocol execution either. A question on how these issues may be resolved posted to the official ProVerif mailing list did not result in further insights until the date of submission. Hence, the only result provided for queries 8, 9, and 10 is the textual representation of the attack trace and the raw dot files.


```
top - 11:55:07 up 1 day, 12:26,  1 user,  load average: 2.04, 2.02, 2.00
Tasks:  92 total,   3 running,  89 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.7 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.3 hi,  0.0 si,  0.0 st
MiB Mem :  31538.3 total,  30625.2 free,    506.0 used,    407.1 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  30647.3 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 5019 ec2-user  20   0  215344 104580  10688 R  99.7   0.3 364:31.36 dot
 5027 ec2-user  20   0  220472 109576  10572 R  99.7   0.3 364:05.66 dot
```

Figure 4.1: Graphviz dot utility execution

Since dot is the only supported output for a visual representation of the discovered attack traces, ProVerif is necessarily limited by the issues with the handling of this file format. The lack of such visualization severely complicates validation of the discovered attacks. It should be noted that ProVerif still prints a text-based description of the attack trace on the standard output. However, given the large amounts of system states this representation is quickly becoming unmanagable for complex protocols itself.

Fourthly, unlike in many popular programming languages, the behavior of constructors and destructors within an if-then-else clause is such that they do not produce a boolean negative return value in case of failure, but the special value fail. This can easily lead to unintended behavior as neither of the branches of the logic tree are executed and instead the process as a whole fails. What is more, ProVerif's output does not contain any warnings about failed constructors, destructors, and processes. Listing 4.11 shows a possible way to prevent this behavior by example of the destructor used to verify signed JSON patches by IPX providers. The definition is structured two parts: the function signature, indicating the input and output types as well as a sequence of several different rewrite rules. The first one indicates the positive case, including a well-formed JSON patch object that has been signed with the same private key as the one belonging to the public key the signature is verified with. In this case, the return value is true. If the definition of the deconstructor would finish at this point, the different possible failures would not be catered for. Hence, the next line starting with the keyword otherwise captures scenarios in which the private/public key pair is not matching and the return value is false. The last otherwise branch is a catch-all rule, capturing cases in which the first input parameter does not match the expected structure of a JSON patch.

```
fun validPrinsSign(bitstring, pubkey): bool
reduc forall p: prins, ops: bitstring, id: bitstring, tag: mac, k: privkey;
    validPrinsSign(signPrins(p, ops, id, tag, k), pk(k)) = true
otherwise forall p: prins, ops: bitstring, id: bitstring, tag: mac,
    k: privkey, k': privkey;
    validPrinsSign(signPrins(p, ops, id, tag, k'), pk(k)) = false
otherwise forall k: privkey;
    validPrinsSign(fail, pk(k)) = false.
```

Listing 4.11: Definition of deconstructors for patch validation

Lastly, ProVerif's lack of support for multi-threaded execution can become a bottleneck when verifying larger systems. Given that the software only ever makes use of a single thread at once, each query is executed sequentially and a blocking query will block all remain ones indefinitely. This is especially problematic when the protocol under test has reached a sizable complexity and existing queries are modified or new queries are added in later stages.

In these situations, splitting up the set of queries into different ProVerif files lends itself as a crude method of macro-parallelization. While effective, this practice is tedious and time-consuming. Ideally, ProVerif could add a command line option that would allow to distribute queries across several threads. Likewise, a simple optimization related to the generation of attack traces explained above, would be spawning the DOT process in parallel to ProVerif itself. That way, in contrast to the current implementation, the time-consuming rendering of these graphs would not block the verification of the remaining queries.

All in all, the above issues related to the reporting of warnings and internal progress during verification, program non-termination, and the visualization of discovered attack traces render the obtained results vague at best. Without a practical way to retrace the executed procotol flow, verifying ProVerif's findings, let alone deriving improvements for the protocol based on them, is simply unfeasible. The program's interactive mode is aimed at addressing this requirement, but handicaped by limited feature support as well, since it only allows structuring protocols using phases, not synchronozation groups which are used in the created model. However, it has to be acknowledged that the underlying problem, that is verifying the models correctness in terms of accurately representing the specification, is something that the tool itself cannot provide unless the specification is not written in a formal way in the first place. This is not the case with the 5G security specification TS 33.501.

4.3 Specification Gaps

Based on the lessons learned during modeling and formally verifying of the PRINS protocol, a number of gaps and inconsistencies in the 3GPP specification have been identified. When addressed, these points can facilitate a more coherent specification, a reduction of potential ambiguity during implementation, and by that an overall sounder protocol.

One of the first observations when analyzing the specification TS 33.501 is the absence of an explicitly defined threat model and security requirements derived from it. Previous research on another component of the 5G system highlighted this as an issue (see Basin, Cremers, and Meadows 2018) and it holds true for the decription of the inter-operator protocol as well. In fact, merely half of the security properties outlined in section 2.1.4 are explicitly spelled out in 3GPP's document. The lack of a clear description of what properties the protocol is supposed to achieve not only impedes the creation of related queries for a formal verification, it is also likely to cause issues during implementation.

Further, the specification makes assumptions about certain information being agreed and correctly configured between protocol participants prior to session establishment without detailing the intended behavior if any of these assumptions is not met. As highlighted in section 2.1.3, it fails to describe if and how the N32-c session establishment is to proceed if either of the data-type encryption policies violates the protection needs of the information elements to be mandatorily encrypted.

Additionally, 3GPP documents TS 33.501 and TS 29.573 describing the PRINS protocol in different levels of detail are misaligned in several locations.

- While TS 33.501 only refers to an authorized IPX ID (3GPP 2019c, p. 139), TS 29.573 mentions both a authorized IPX ID and a next hop ID in multiple locations (3GPP 2019b, pp. 65-66). Based on their placement within the N32-f message, it can be assumed that these parameters actually refer to the same information, but this connection should be made more clear.

- TS 33.501 specifies that the PLMN ID is supposed to be validated by the receiving SEPP (3GPP 2019c, p. 143). However, it does not detail where in the N32-f message the parameter is transferred. TS 29.573 cites an authorization header containing this information (3GPP 2019b, p. 19). Again, explicitly spelling out this relation would improve the coherence of the two specifications.

- Although TS 33.501 defines a validity parameter as part of the N32-f context information (3GPP 2019c, p. 134), it does not clearly specify to what exactly this parameter applies – the whole context including protection policies or just the cryptographic material.

- Neither TS 33.501 nor TS 29.573 specify the intended behavior of a SEPP once the above-mentioned validity has expired. While the latter contains a description of a N32-f Context Termination Procedure (3GPP 2019b, p. 17), it is unclear whether that is automatically triggered when the end of the context validity is reached or if some parameter renegotiation should be performed instead.

While SEPP vendors would surely be able find a solution to these inconsistencies during implementation, it is not guaranteed that other system suppliers would opt for the same behavior in their components. Given a key purpose of standardization is compatibility between system components, it seems as though the specification of PRINS is still too vague to ensure this.

Chapter 5

Closing Remarks

5.1 Conclusion

Looking back at section 1.2, the work done as part of this thesis successfully manages to address the objectives initially outlined. The three research questions can be answered as follows.

(1) Is it possible to comprehensively describe the 5G inter-operator signaling protocol with existing modeling techniques?

Yes, with a number of abstractions. The high-level comparison between Tamarin and ProVerif shows that both tools should in theory be able to express and validate important aspects of the PRINS protocol. Only a few useful modeling constructs in ProVerif, such as tables, synchronization, and better support for observational equivalence proofs, favor this tool.

The created model successfully incorporates all key characteristics of the PRINS protocol, including multi-channel communication, separation of protocol phases, multiple active participants, intermediary message modifications, and all necessary cryptographic operations (i.e. signatures, MACs, and AEAD encryption). Minor aspects are abstracted away, such as the detailed structure of protection policies, the TLS key export, and the invalidation of N32-f sessions as described in section 3.2. However, given their impact on the security properties to be verified, these do not have an impact on the verification result.

That said, a degree of uncertainty about the model's correctness persists, due to the problems encoutered during verification. Although all debug queries succeed and thereby greatly increase the confidence in the model, for the reasons layed out in section 4.2, there is still a chance ProVerif's execution actually diverges from the intended protocol behavior.

(2) Does modeling PRINS in a formal manner reveal any inconsistencies that should be addressed by the 3GPP specification?

Yes, the previous section contains a summary of several areas that were either ambiguous or completely missing from the specifications TS 33.501 and TS 29.573. Some of these should be amendable by ensuring a uniform naming scheme between the two documents or minor additions of missing procedures. Other aspects, such as a clearly defined threat model and security requirements, are more fundamental in nature and likely require comprehensive revision of the specification. While the authors of the main security specification TS 33.501 might have a comprehensive view of these aspects, they are not explicitly captured in the document or any related 3GPP study documents. Not only does this impede a clear understanding for implementers and mobile operators of what the protocol is supposed to deliver and what might be necessary to address by additional, non-standard controls. But also, it inhibits security research to properly verify these protocols against their intended properties.

The author argues that, given the importance of secure and reliable communication for some of the mission critical use cases 5G is supposed to address, specifying a concrete threat model and security requirements derived from it is indispensable. More so than a formal verification, which proved rather fragile in practice, a precise definition of the requirements would greatly facilitate the development of a secure protocol.

(3) Based on the model created as part of this work, can current formal verification tools prove or disprove particular security requirements of the PRINS protocol?

Yes, partially. Section 4.1 shows that confidentiality protection for all sensitive information is indeed ensured if the protocol is implemented according to the specification. Note, that since the verification is performed in the symbolic domain, the absence of cryptographic flaws such as nonce reuse resulting in a breach of confidentiality with AEAD ciphers, cannot be verified. Moreover, message modifications by unknown IPX providers or modifications not captured in the modification policy are never accepted by the receiving party.

Properties that were disproven include the rejection of replayed message modifications, detection of the removal of previously added message modifications, as well as the integrity protection of JSON patches. While the respective queries leave some margin of error for potential false attacks to be found by ProVerif, as outlined in section 4.1, there is no way to tell without the discovered attack traces available. What is more, the authentication of N32-f messages itself could not be proven, as it resulted in the non-termination of ProVerif.

Based on these results, it would be premature to speak of a vulnerability or weakness in the PRINS protocol itself. Instead, further work is needed in order to validate the created model, refine the verification performed as part of this thesis, and substantiate the findings. A few approaches that may guide future efforts are outlined in the last section.

5.2 Future Work

As shown in section 4.1.4, the work done to model the PRINS protocol for ProVerif as part of this thesis allows for the successful verification of several intended security properties. Nevertheless, there is room for improving and expanding on the formal verification with ProVerif in order to validate protocol aspects that have not been covered conclusively.

Firstly, tuning of the resolution strategy employed by ProVerif to derive facts from available clauses may speed-up scenarios that did not terminate using the available default strategies. The program's user manual describes how facts can be assigned a custom weight (Blanchet, Smyth, et al. 2020, pp. 112-114), guiding the resolution on what facts to prefer or ignore. While it is not guaranteed that this optimization alone would resolve the issue of non-termination for the concerned queries, it is a refinement step in the verification that has not been tried so far.

Secondly, further analyzing the attack traces produced by ProVerif for the queries that have been explicitly disproven would likely yield additional insights based on the work already performed. Whether it is optimizing the generation of visual attack graphs using a different toolset or manually retracing the text-based description generated by ProVerif, these steps not performed due to the time constraints of this thesis could help getting the most out of the results obtained so far and potentially guide further model improvements or even protocol amendments.

Lastly, certain existing security queries could be refined to produce more meaningful results. The latest ProVerif version 2.02 released in July 2020 adds support for specifying time points in correspondence assertions which could be used to more accurately verify some of the queries in section 4.1. For example, the ones involving rogue IPX providers and subsequent validation of their message modifications could benefit from the ability to express temporal relations to either confirm or clear up the discovery of a practical attack. The only reason why verification has not been attempted with this newer version of the software is that ProVerif 2.02 introduces breaking changes that lead to the non-termination of further queries.

These points show that there are several potential ways to directly build on the work done as part of this thesis. Beyond that, approaching the verification of PRINS from another angle, for example modeling it for a completely different tool may yield additional insights. Rather than a conclusive analysis, this thesis should instead be seen as a first step in verifying the security of PRINS. Ideally, it can contribute to a greater awareness of this protocol within the security research community and by that, the security of the 5G system as a whole.

Literature

3GPP (June 2019a). *3G security; Network Domain Security (NDS); IP network layer security.* Technical Specification (TS) 33.210. Version 16.2.0. 3rd Generation Partnership Project (3GPP).

— (Dec. 2019b). *5G System; Public Land Mobile Network (PLMN) Interconnection; Stage 3.* Technical Specification (TS) 29.573. Version 16.1.0. 3rd Generation Partnership Project (3GPP).

— (Dec. 2019c). *Security architecture and procedures for 5G System.* Technical Specification (TS) 33.501. Version 16.1.0. 3rd Generation Partnership Project (3GPP).

Abadi, Martin, Bruno Blanchet, and Cédric Fournet (2017). "The applied pi calculus: Mobile values, new names, and secure communication". In: *Journal of the ACM (JACM)* 65.1, pp. 1–41.

Alt, Stephanie et al. (2016). "A Cryptographic Analysis of UMTS/LTE AKA". In: *International Conference on Applied Cryptography and Network Security.* Springer, pp. 18–35.

Alur, Rajeev (2011). "Formal verification of hybrid systems". In: *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT).* IEEE, pp. 273–278.

Alur, Rajeev and Mihalis Yannakakis (1998). "Model checking of hierarchical state machines". In: *ACM SIGSOFT Software Engineering Notes* 23.6, pp. 175–188.

Baeten, JCM, Dirk Albert van Beek, and JE Rooda (2007). "Process algebra". In: *Handbook of Dynamic System Modeling.*

Baier, Christel and Joost-Pieter Katoen (2008). *Principles of model checking.* MIT press.

Basin, David, Cas Cremers, and Catherine Meadows (2018). "Model checking security protocols". In: *Handbook of Model Checking.* Springer, pp. 727–762.

Basin, David, Jannik Dreier, Lucca Hirschi, et al. (2018). "A formal analysis of 5G authentication". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1383–1396.

Basin, David, Jannik Dreier, and Ralf Sasse (2015). "Automated symbolic proofs of observational equivalence". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1144–1155.

Blanchet, Bruno (2012). "A computationally sound mechanized prover for security protocols". In: *IEEE Transactions on Dependable and Secure Computing* 5.4, pp. 193–207.

— (2013). "Automatic verification of security protocols in the symbolic model: The verifier proverif". In: *Foundations of Security Analysis and Design VII.* Springer, pp. 54–87.

Blanchet, Bruno et al. (2016). “Modeling and verifying security protocols with the applied pi calculus and ProVerif”. In: *Foundations and Trends in Privacy and Security* 1.1-2, pp. 1–135.

Blanchet, Bruno (2017). “Symbolic and computational mechanized verification of the ARINC823 avionic protocols”. In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, pp. 68–82.

Blanchet, Bruno, Aaron D. Jaggard, et al. (Mar. 2008). “Computationally Sound Mechanized Proofs for Basic and Public-key Kerberos”. In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*. Tokyo, Japan: ACM, pp. 87–99.

Blanchet, Bruno, Ben Smyth, et al. (2020). “ProVerif 2.01: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial”. In:

Borgaonkar, Ravishankar et al. (2019). “New privacy threat on 3G, 4G, and upcoming 5G AKA protocols”. In: *Proceedings on Privacy Enhancing Technologies* 2019.3, pp. 108–127.

Burch, Jerry R et al. (1992). “Symbolic model checking: 1020 states and beyond”. In: *Information and computation* 98.2, pp. 142–170.

Cadé, David and Bruno Blanchet (Mar. 2013). “From Computationally-Proved Protocol Specifications to Implementations and Application to SSH”. In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 4.1. Special issue ARES'12, pp. 4–31.

Clarke, Edmund M. and E. Allen Emerson (1981). “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Workshop on Logic of Programs*. Springer, pp. 52–71.

Cremers, Cas (2008). “The Scyther Tool: Verification, falsification, and analysis of security protocols”. In: *International conference on computer aided verification*. Springer, pp. 414–418.

Cremers, Cas et al. (2017). “A comprehensive symbolic analysis of TLS 1.3”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1773–1788.

Dehnel-Wild, Martin and Cas Cremers (2018). “Security vulnerability in 5G-AKA draft”. In: *Department of Computer Science, University of Oxford, Tech. Rep.*

Dolev, Danny and Andrew Yao (1983). “On the security of public key protocols”. In: *IEEE Transactions on information theory* 29.2, pp. 198–208.

Dreier, Jannik et al. (2017). “Beyond subterm-convergent equational theories in automated verification of stateful protocols”. In: *International Conference on Principles of Security and Trust*. Springer, pp. 117–140.

Emerson, E. Allen (1990). “Temporal and modal logic”. In: *Formal Models and Semantics*. Elsevier, pp. 995–1072.

ENISA (Mar. 2018). *Signalling Security in Telecom SS7/Diameter/5G*. Tech. rep. European Union Agency for Network and Information Security (ENISA).

Etessami, Kousha and Gerard J Holzmann (2000). “Optimizing büchi automata”. In: *International Conference on Concurrency Theory*. Springer, pp. 153–168.

Fersch, Manuel, Eike Kiltz, and Bertram Poettering (2016). “On the provable security of (EC) DSA signatures”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1651–1662.

Fielding, Roy T. and Richard N. Taylor (2000). *Architectural styles and the design of network-based software architectures.* Vol. 7. University of California, Irvine Irvine.

Hirschi, Lucca and Cas Cremers (2019). “Improving automated symbolic analysis of ballot secrecy for e-voting protocols: A method based on sufficient conditions”. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE, pp. 635–650.

Holtmanns, Silke and Ian Oliver (May 2017). “SMS and one-time-password interception in LTE networks”. In: *2017 IEEE International Conference on Communications (ICC)*, pp. 1–6.

Holtmanns, Silke, Siddharth Prakash Rao, and Ian Oliver (2016). “User location tracking attacks for LTE networks using the interworking functionality”. In: *2016 IFIP Networking conference (IFIP Networking) and workshops.* IEEE, pp. 315–322.

Horvat, Marko (2015). “Formal analysis of modern security protocols in current standards”. PhD thesis. University of Oxford.

Kim, Jun Young et al. (2017). “Automated analysis of secure internet of things protocols”. In: *Proceedings of the 33rd Annual Computer Security Applications Conference*, pp. 238–249.

Kobeissi, Nadim, Karthikeyan Bhargavan, and Bruno Blanchet (2017). “Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach”. In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE, pp. 435–450.

Kremer, Steve and Robert Künnemann (2016). “Automated analysis of security protocols with global state”. In: *Journal of Computer Security* 24.5, pp. 583–616.

LaMacchia, Brian, Kristin Lauter, and Anton Mityagin (2007). “Stronger security of authenticated key exchange”. In: *International conference on provable security.* Springer, pp. 1–16.

Lee, Ming-Feng et al. (2014). “Anonymity guarantees of the UMTS/LTE authentication and connection protocol”. In: *International journal of information security* 13.6, pp. 513–527.

Lipp, Benjamin, Bruno Blanchet, and Karthikeyan Bhargavan (June 2019). “A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol”. In: *IEEE European Symposium on Security and Privacy (EuroS&P’19).* Stockholm, Sweden: IEEE Computer Society, pp. 231–246.

Lowe, Gavin (1996). “Breaking and fixing the Needham-Schroeder public-key protocol using FDR”. In: *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, pp. 147–166.

— (1997). “A hierarchy of authentication specifications”. In: *Proceedings 10th Computer Security Foundations Workshop.* IEEE, pp. 31–43.

Mayer, Georg (2018). “RESTful APIs for the 5G service based architecture”. In: *Journal of ICT Standardization* 6.1, pp. 101–116.

McGrew, David A and John Viega (2004). “The security and performance of the Galois/-Counter Mode (GCM) of operation”. In: *International Conference on Cryptology in India*. Springer, pp. 343–355.

Meier, Simon (2013). “Advancing automated security protocol verification”. PhD thesis. ETH Zurich.

Meier, Simon et al. (2013). “The TAMARIN prover for the symbolic analysis of security protocols”. In: *International Conference on Computer Aided Verification*. Springer, pp. 696–701.

Merwe, Thyla van der (2018). “An Analysis of the Transport Layer Security Protocol”. PhD thesis. Royal Holloway, University of London.

Milner, Robin (1999). *Communicating and mobile systems: the pi calculus*. Cambridge university press.

Needham, Roger M. and Michael D. Schroeder (1978). “Using encryption for authentication in large networks of computers”. In: *Communications of the ACM* 21.12, pp. 993–999.

NGMN Alliance (2015). *5G White Paper*.

Rao, Siddharth Prakash, Silke Holtmanns, et al. (Aug. 2015). “Unblocking Stolen Mobile Devices Using SS7-MAP Vulnerabilities: Exploiting the Relationship between IMEI and IMSI for EIR Access”. In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1, pp. 1171–1176.

Rao, Siddharth Prakash, Bhanu Teja Kotte, and Silke Holtmanns (2016). “Privacy in LTE networks”. In: *Proceedings of the 9th EAI International Conference on Mobile Multimedia Communications*, pp. 176–183.

Rao, Siddharth Prakash, Ian Oliver, et al. (2016). “We know where you are!” In: *2016 8th International Conference on Cyber Conflict (CyCon)*. IEEE, pp. 277–293.

Schmidt, Benedikt (2012). “Formal analysis of key exchange protocols and physical protocols”. PhD thesis. ETH Zurich.

Schöning, Uwe (2008). *Logic for computer scientists*. Springer Science & Business Media.

The Tamarin Team (June 2019). “Tamarin-Prover Manual”. In:

Vodafone (2018). *pCR to TS33.501 - Session binding to prevent potential 5G-AKA vulnerability*. Tech. rep. S3-180727. 3GPP TSG SA WG3 Meeting 90Bis. Vodafone GmbH.

Online Sources

Basin, David, Cas Cremers, Jannik Dreier, et al. (2020). *Tamarin Prover.* URL: `https://tamarin-prover.github.io/` (visited on 01/26/2020).

Blanchet, Bruno (2020). *ProVerif: Cryptographic protocol verifier in the formal model.* URL: `https://prosecco.gforge.inria.fr/personal/bblanche/proverif/` (visited on 01/26/2020).

Bryan, P. and M. Nottingham (2013). *RFC 6902.* JavaScript Object Notation (JSON) Patch. URL: `https://tools.ietf.org/html/rfc6902` (visited on 02/18/2020).

Cremers, Cas (2020). *The Scyther Tool.* URL: `https://people.cispa.io/cas.cremers/scyther/` (visited on 01/26/2020).

De Oliveira, Alexandre et al. (2014). *Worldwide attacks on SS7 network.* Hackito Ergo Summit. URL: `http://2014.hackitoergosum.org/slides/day3_Worldwide_attacks_on_SS7_network_P1security_Hackito_2014.pdf` (visited on 01/26/2020).

Engel, Tobias (2008). *Locating Mobile Phones using Signaling System 7.* 25th Chaos Communication Congress 25C3. URL: `http://berlin.ccc.de/~tobias/25c3-locating-mobile-phones.pdf` (visited on 01/26/2020).

— (2014). *SS7: Locate. Track. Manipulate.* 31st Chaos Communication Congress 31C3. URL: `http://berlin.ccc.de/~tobias/31c3-ss7-locate-track-manipulate.pdf` (visited on 01/26/2020).

Puzankov, Sergey and Dmitry Kurbatov (2014). *How to Intercept a Conversation Held on the Other Side of the Planet.* PHDays. URL: `http://2014.phdays.com/program/tech/36930/` (visited on 01/26/2020).

Rescorla, E. (2010). *RFC 5705.* Keying Material Exporters for Transport Layer Security (TLS). URL: `https://tools.ietf.org/html/rfc5705` (visited on 02/29/2020).

Appendix A

ProVerif Model and Results

The ProVerif model is not printed here verbatim due to its large size. The complete source code, including the discovered attack traces in plaintext and dot format, can be retreived under the following link: `https://github.com/hcrudolph/proverif-prins`.