Automated security verification of networking protocols and query auditing algorithms for wireless sensor networks

Ph.D. Dissertation of Ta Vinh Thong

Supervisor: Levente Buttyán, Ph.D.

Budapest, Hungary 2013
Alulírott Ta Vinh Thong kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szereint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

I, the undersigned Ta Vinh Thong hereby declare, that this Ph.D. dissertation was made by myself, and I only used the sources given at the end. Every part that was quoted word-for-word, or was taken over with the same content, I noted explicitly by giving the reference of the source.

A dolgozat birkálatai és a védésről készült jegyzőkönyv a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Dekán hivatalában elérhetőek.

The reviews of the dissertation and the report of the thesis discussion are available at the Dean’s Office of the Faculty of Electrical Engineering and Informatics of the Budapest University of Technology and Economics.

Budapest, ....................

_________________________
Ta Vinh Thong
Abstract

Wireless Sensor Networks (WSNs) are given higher priority recently, thanks to their increasingly important role and widespread applications in everyday life. WSNs consist of spatially distributed sensors (called sensor nodes) to monitor physical or environmental conditions, such as temperature, sound, pressure, etc., at different locations. Each sensor node typically has a radio transceiver with an internal antenna or connection to an external antenna, a microcontroller, an electronic circuit for interfacing with the sensors and an energy source, usually a battery. WSNs consist of a large number of resource constrained sensor nodes and a few more powerful base stations. The sensors collect various types of data from the environment and send those data to the base stations using multi-hop wireless communications. For this reason, in the literature, the base stations are also called sink nodes. Communications in WSNs usually take place between the sensor nodes and the base stations, and it is important to distinguish the direction of those communications. In case of upstream communication, the sender is a sensor node, and the receiver is a base station, while in case of downstream communication, these roles are reversed. The goal of the sender is to reliably transmit to the receiver a full message that may consist of multiple fragments.

The importance of WSNs arises from their capability for detailed monitoring in remote and inaccessible locations where it is not feasible to install conventional wired infrastructure. The development of WSNs was motivated by military applications such as battlefield surveillance, however, today such networks are used in many industrial and consumer applications, including medical monitoring [65], [32], environmental monitoring [69], air pollution monitoring [41], natural disaster prevention, forest fire detection, smart home monitoring, and industrial machine monitoring [8]. There are a number of research challenges associated with wireless sensor communication arising from the limited capabilities of low cost sensor node hardware and the common requirement for nodes to operate for long time periods with only a small battery. For this reason, most security solutions designed for wired networks, which often include memory and computation intensive tasks, cannot be directly adopted in wireless sensor networks.

Up to date, numerous networking protocols and solutions have been proposed to ensure the reliable operation of WSNs applications in a hostile environment [19]. However, despite the fact that WSNs are often envisioned to operate in hostile environments, some of the protocols and solutions do not address security issues at all, and as a consequence they ensure reliability only in a benign environment where no intentional attack takes place. Recognizing this problem, in recent years many research focused on proposing security protocols based on cryptographic methods [19]. Unfortunately, designing security protocols is a very difficult and error-prone task, as confirmed by the fact that critical security holes can be found in many widely used protocols, including protocols secured by cryptographic operations, and believed to be secure by the protocol designers. The security vulnerabilities inherent in the designed protocols are often hard to spot, because of the huge number of behavioral scenarios defined in the protocols. In many cases, protocol and system designers only perform manual and informal analysis on their proposed protocols. The main problem is that informal analysis of protocols is error-prone, and security holes can be overlooked, hence, it is not considered to be a reliable approach. Addressing this problem, my research focuses
on formal analysis and automated security verification of protocols for wireless sensor networks. Formal analysis is based on strong mathematical background, and uses formal languages that have expressive syntax and semantics, and give us a possibility to automate the security verification.

Within this area of research, my work concentrates on investigating the security problems in different WSN and ad-hoc networks applications, namely, the security of (1) wireless ad-hoc networks routing protocols, (2) transport protocols designed for WSNs, and (3) the application of WSNs in health-care management. Each of these three topics has received a significant amount of research attention, and many related papers have been published (see e.g., [54], [18], [3], [36], [35], [16], [47], [5], [21]).

Ad-hoc networks are not based on predefined topology, thus, in order to allow one node to communicate with another node, route discovery is accomplished based on routing protocols. Due to the design flaws of routing protocols, numerous route forging attacks against routing protocols have been published, in which attacker(s) can achieve that the honest parties attempt to exchange data through a route that does not exist in reality, without being aware of it [18], [3]. This type of attacks is critical because they can lead to futile energy consumption and degrade the efficiency of the network. Several works investigated formal methods to analyze routing protocols, in order to detect vulnerabilities can be found in them. These methods have many drawbacks, since they are based on formal languages that lack required modeling elements, and are non-automated.

In the second topic, some WSN applications require the use of a transport protocol that ensures reliable delivery and congestion control. However, despite the fact that WSNs are often envisioned to operate in hostile environments, existing transport protocols for WSNs do not address security issues at all and as a consequence, many attacks have been detected against well-known WSN transport protocols [16]. Up to date, WSNs transport protocols have been only analyzed informally and manually by the designers, and there is not any formal and automated verification method proposed for this class of protocols so far.

As for the third topic, in remote patient monitoring applications, sensor readings are collected on personal mobile device, such as a mobile phone. Third parties can then access these database by sending queries to the mobile device. For this kind of application it is crucial to preserve the privacy of the patients, and sensitive information about their health status must not be obtainable by unauthorized parties. Hence, proposing query auditing methods that prevent and detect the disclosure of sensitive information are indispensable, and have been extensively investigated [5].

In my dissertation, I propose formal and automated verification methods for analyzing the security of routing and transport protocols, as well as proposing methods for protecting sensitive database information collected from sensor devices. The dissertation is composed of three thesis groups which are related to the following three research topics: (1) formal and automated security analysis of routing protocols for wireless ad-hoc sensor networks; (2) formal and automated verification of transport protocols for wireless sensor networks; and (3) query auditing algorithms for protecting sensitive information in statistical databases.
Acknowledgement

First of all, I would like to express my gratitude to my supervisor, Professor Levente Buttyán, Ph.D., Departement of Networked Systems and Services, Budapest University of Technology and Economics. Thank you very much for giving me the opportunity to work in the fantastic and prestigious CrySyS Lab. Thank you for your guidance in selecting problems to work on, your advices in elaborating the problems and in publishing the results. All these three steps were needed to finish this dissertation.

I am also grateful to the current and former members of the CrySyS Laboratory: Gergely Ács, Boldizsár Bencsáth, László Czap, László Csók, László Dóra, Amit Dvir, Márk Félegyházi, Gábor Gulyás, Tamás Holczer, Gergely Kötyuk, Áron Lászka, Gábor Péter, Péter Schaffer, and István Vajda for the illuminating discussions on different technical problems that I encountered during my research. Thank you for providing a pleasant atmosphere which was a pleasure to work in.

I would also like to thank for our joint efforts and publications to Gorgio Calandriello, Amit Dvir, Albert Held, Jean-Pierre Hubaux, Frank Kargl, Antonio Kung, Michael Müter, Panagiotis Papadimitratos, Elmar Schoch, and Björn Wiedersheim.

I am thankful to the reviewers of my thesis, István Majzik and Panos Papadimitratos for the fast and careful reviews, and the large number of useful comments. I am grateful that they reviewed and evaluated my thesis in spite of the limited time that they had.

The financial support of the Ericsson via the HSNLab at BME, the grant TAMOP - 4.2.2.B-10/12010-0009 and the support of the SEVECOM (FP6-027795), the WSAN4CIP (FP7-225186), and the CHIRON (ARTEMIS 225186) EU projects are gratefully acknowledged.

And last but not least I want to thank my family and best friends for always supporting me. This thesis would not have been possible without their support.
# Contents

1 Introduction

2 Formal and automated security verification of wireless ad-hoc routing protocols
   2.1 Introduction ................................................. 5
   2.2 Route forging attacks against secure routing protocols ............... 6
      2.2.1 The SRP protocol ..................................... 6
      2.2.2 The Ariadne protocol ................................. 7
      2.2.3 The endairA protocol ................................. 9
      2.2.4 Summary ............................................. 10
   2.3 Related works ............................................ 10
      2.3.1 Related formal analysis methods ........................ 10
      2.3.2 Related automatic verification methods ....................... 11
   2.4 Assumptions on routing protocols and the attacker model ............. 12
   2.5 The proposed sr-calculus .................................. 13
      2.5.1 Type system of the sr-calculus ........................ 13
      2.5.2 Formal Syntax of the sr-calculus ........................ 14
      2.5.3 Operational Semantics of the sr-calculus ................... 16
      2.5.4 Attacker knowledge base, static equivalence, labeled bisimilarity . 17
      2.5.5 Examples ........................................... 20
   2.6 Analyzing the SRP protocol with the sr-calculus ....................... 22
   2.7 BDSR: The backward deduction algorithm for source routing protocols . 24
      2.7.1 Applying BDSR for the sr-calculus ....................... 27
      2.7.2 General specification of on-demand source routing protocols ....... 29
      2.7.3 Analyzing the security of Ariadne based on the BDSR algorithm 30
      2.7.4 Analyzing the security of endairA based on the BDSR algorithm 36
   2.8 sr-verif: On automating the verification .......................... 40
      2.8.1 The concept of the verification method ........................ 40
      2.8.2 Specifying on-demand source routing protocols ................... 42
      2.8.3 From protocol specification to logic rules ....................... 45
      2.8.4 Syntax of the logic rules ................................ 45
      2.8.5 The resulting protocol rules ................................ 46
      2.8.6 Specifying the attacker rules ............................... 50
      2.8.7 Automating the verification using resolution-based deduction and backward  searching .................................. 51
      2.8.8 Derivation ........................................... 51
      2.8.9 The verification algorithm: Combining BDSR with logic based resolution . 53
      2.8.10 Termination .......................................... 59
      2.8.11 Correctness and completeness .............................. 62
Chapter 1

Introduction

Wireless Sensor Networks (WSNs) are given higher priority recently, thanks to their increasingly important role and widespread applications in everyday life. WSNs consist of spatially distributed sensors (called sensor nodes) to monitor physical or environmental conditions, such as temperature, sound, pressure, etc., at different locations. Each sensor node typically has a radio transceiver with an internal antenna or connection to an external antenna, a microcontroller, an electronic circuit for interfacing with the sensors and an energy source, usually a battery. WSNs consist of a large number of resource constrained sensor nodes and a few more powerful base stations. The sensors collect various types of data from the environment and send those data to the base stations using multi-hop wireless communications. For this reason, in the literature, the base stations are also called sink nodes. Communications in WSNs usually take place between the sensor nodes and the base stations, and it is important to distinguish the direction of those communications. In case of upstream communication, the sender is a sensor node, and the receiver is a base station, while in case of downstream communication, these roles are reversed. The goal of the sender is to reliably transmit to the receiver a full message that may consist of multiple fragments.

The importance of WSNs arises from their capability for detailed monitoring in remote and inaccessible locations where it is not feasible to install conventional wired infrastructure. The development of WSNs was motivated by military applications such as battlefield surveillance, however, today such networks are used in many industrial and consumer applications, including medical monitoring [65], [32], environmental monitoring [69], air pollution monitoring [41], natural disaster prevention, forest fire detection, smart home monitoring, and industrial machine monitoring [8]. There are a number of research challenges associated with wireless sensor communication arising from the limited capabilities of low cost sensor node hardware and the common requirement for nodes to operate for long time periods with only a small battery. For this reason, most security solutions designed for wired networks, which often include memory and computation intensive tasks, cannot be directly adopted in wireless sensor networks.

Up to date, numerous networking protocols and solutions have been proposed to ensure the reliable operation of WSNs applications in a hostile environment [19]. However, despite the fact that WSNs are often envisioned to operate in hostile environments, some of the protocols and solutions do not address security issues at all, and as a consequence they ensure reliability only in a benign environment where no intentional attack takes place. Recognizing this problem, in recent years many research focused on proposing security protocols based on cryptographic methods [19]. Unfortunately, designing security protocols is a very difficult and error-prone task, as confirmed by the fact that critical security holes can be found in many widely used protocols, including protocols secured by cryptographic operations, and believed to be secure by the protocol designers. The security vulnerabilities inherent in the designed protocols are often hard to spot, because of the huge number of behavioral scenarios defined in the protocols. In many cases, protocol and system designers only perform manual and informal analysis on their proposed protocols. The main
1. INTRODUCTION

problem is that informal analysis of protocols is error-prone, and security holes can be overlooked, hence, it is not considered to be a reliable approach. Addressing this problem, my research focuses on formal analysis and automated security verification of protocols for wireless sensor networks. Formal analysis is based on strong mathematical background, and uses formal languages that have expressive syntax and semantics, and give us a possibility to automate the security verification.

In this dissertation, I propose formal and automated verification methods for analysing the security of protocols. I focus on the protocols and algorithms designed for wireless sensor and wireless ad-hoc networks, which are related to the following three topics: (1) formal and automated security analysis of routing protocols for wireless ad-hoc networks; (2) formal and automated verification of transport protocols for wireless sensor networks (WSNs); and (3) query auditing algorithms for protecting sensitive information in statistical databases. In the following, I provide a brief overview of the three research topics that are covered in my dissertation.

**Topic 1:** My first topic, discussed in Chapter 2, is related to a more general form of WSNs. Namely, I focus on such applications in which the sensor nodes are deployed in devices which permanently change their locations, such as vehicular networks. These networks are known as wireless ad-hoc networks, in which unlike WSNs, any node can be the source and the target. Wireless ad-hoc networks are not based on pre-defined topology, thus, in order to allow one party to communicate with another party, route discovery is accomplished. Once a route between two parties has been found, they start to exchange data on this route such that each party in the route forward the packet it received to the target. The route discovery procedure is defined by routing protocols. Numerous attacks against routing protocols have been published, in which attacker(s) can achieve that the honest parties attempt to exchange data through a route that does not exist in reality, without being aware of it. This type of attacks is critical because it can lead to futile energy consumption and degrade the efficiency of the network.

**My contributions:** I propose a process algebra variant that, unlike previous works, provides expressive syntax and semantics for analyzing at the same time (i.) cryptographic primitives and operations, (ii.) the nature of broadcast communication, and (iii.) the specification of node’s neighborhood, which are required for verifying secure routing protocols. In addition, the main challenge of automated analysis of routing protocols is that, during the verification, a large number of network topologies and a strong attacker model need to be considered. This induces a huge number of states to be examined, which a computer cannot handle. To overcome this, I propose a novel automated verification method that is able to handle arbitrary network topologies and a strong attacker model, which previous methods cannot provide.

**Topic 2:** The second topic, detailed in Chapter 3, is concerned with the security verification of transport protocols designed for wireless sensor networks. In some applications of WSNs, for instance, in case of multimedia sensor networks [6], the sensors capture and transmit high-rate data with some QoS requirements. Such applications require the use of a transport protocol that ensures reliable delivery and congestion control. Transport protocols used in wired networks (e.g., the well-known TCP) are not applicable in WSNs, because they perform poorly in a wireless environment and they are not optimized for energy consumption. Therefore, a number of transport protocols specifically designed for WSNs have been proposed in the literature (see e.g., [72] for a survey). The main design criteria that those transport protocols try to meet are reliability and energy efficiency. Unfortunately, existing transport protocols for WSNs do not include sufficient security mechanisms or totally ignore security. Hence, many attacks have been found against existing WSN transport protocols. In general, we can talk about attacks against reliability and energy depleting attacks. An attack against reliability is considered to be successful if the loss of a packet (or packet fragment) remains undetected. In case of energy depleting attacks, the goal of the attacker is to force the sensor nodes to perform energy intensive operations, in order to deplete their batteries.

**My contributions:** I propose solutions for analyzing and verifying the security of WSN transport protocols. The verification of this class of protocols is difficult because they typically consist of complex behavioral characteristics, such as real-time, probabilistic, and cryptographic operations. To solve this problem, I propose a probabilistic timed calculus for cryptographic protocols, and demonstrate how to use this formal language for proving security or vulnerability
of protocols. To the best of my knowledge, this is the first such process calculus that supports an expressive syntax and semantics, real-time, probabilistic, and cryptographic issues at the same time. Hence, it can be used to verify systems that involve these three properties. In addition, I propose an automatic verification method, based on the well-known PAT process analysis toolkit, for this class of protocols. For demonstration purposes, I apply the proposed manual and automatic proof methods for verifying the security of DTSN and SDTP, which are two of the proposed WSN transport protocols. I also proposed a new secured WSN transport protocol, SDTP+, and analyzed its security based on my proposed formal method.

**Topic 3:** My third research topic is discussed in Chapter 4, and it focuses on the application of WSNs in hospital environment, where body mounted wireless sensor networks are used to collect medical data (e.g., ECG signals, blood pressure measurements, temperature samples, etc.) from a patient, and a personal device (e.g., a smart phone) is used to collect those data. The measured records are stored in a database on the personal device, and in the most cases they are sensitive information that only authorized person (e.g., attending physician) can access. In many cases, access to some kind of statistical information about the stored data is allowed to external parties (e.g., hospital personnel, personal coach services, and health insurance companies, researchers). The statistical data is not sensitive for the patient, and one important requirement is that from the set of statistical data, the sensitive information cannot be inferred. For instance, the queries about the average of sensitive data are allowed to be provided, however, from these averages individual sensitive measurement data samples should not be deducible. To achieve this, the so called query auditors are deployed in the personnel devices.

Query auditing (QA) is the problem that has been studied intensively in the context of disclosure control in statistical databases. The goal of an off-line query auditing algorithm is to decide whether private information was disclosed by the responses of the database to a certain set of aggregate queries. Off-line query auditors work on queries received and responses provided in the past, therefore, they can only detect a privacy breach, but cannot prevent it. On-line query auditing algorithms, on the other hand, decide whether responding to a new incoming query would result in the disclosure of some private information, given the responses that have already been provided to past queries, and if responding to the new query would breach privacy, then the database can deny the response. Thus, on-line query auditing algorithms can prevent the unintended disclosure of private information. Various disclosure models are considered, namely, full disclosure and partial disclosure models. In the full disclosure case, the privacy of some data \( x \) breaches when \( x \) has been uniquely determined, while in the latter case \( x \) has been inferred to fall in a set consisting only small number of the possible values.

**My contributions:** I define a novel variant for query auditing, where instead of detecting or preventing the disclosure of individual sensitive values, I want to detect or prevent the disclosure of aggregate values in the database. I study the problem of detecting or preventing the disclosure of the maximum value in the database, when the querier is allowed to issue average queries to the database, because some of those aggregates (extreme values) can be used to infer the health status of the patient. I propose efficient off-line and on-line query auditors for this problem in the full disclosure model, and a simulatable on-line query auditor in the partial disclosure model.
Chapter 2

Formal and automated security verification of wireless ad-hoc routing protocols

2.1 Introduction

In the recent past, the idea of ad-hoc networks have created a lot of interest in the research community, and it is now starting to materialize in practice in various forms, ranging from static sensor networks through opportunistic interactions between personal communication devices to vehicular networks with increased mobility. A common property of these systems is that they have sporadic access, if at all, to fixed, pre-installed communication infrastructures. Hence, it is usually assumed that the devices in ad-hoc networks play multiple roles: they are terminals and network nodes at the same time. Ad-hoc networks do not rely on a pre-defined network topology, hence, before communicating route discovery between two nodes is performed.

In their role as network nodes, the devices in ad-hoc networks perform basic networking functions, most notably routing. At the same time, in their role as terminals, they are in the hands of end-users, or they are installed in physically easily accessible places. In any case, they can be easily compromised and re-programmed such that they do not follow the routing protocol faithfully. The motivations for such re-programming could range from malicious objectives (e.g., to disrupt the operation of the network) to selfishness (e.g., to save precious resources such as battery power). The problem is that such compromised and misbehaving routers may have a profound negative effect on the performance of the network.

In order to mitigate the effect of misbehaving routers on network performance, a number of secured routing protocols have been proposed for ad-hoc networks (see e.g., [36] for a survey). These protocols use various mechanisms, such as cryptographic coding, multi-path routing, and anomaly detection techniques, to increase the resistance of the protocol to attacks. Unfortunately, the design of secure routing protocols is an error-prone activity, and indeed, most of the proposed secure ad-hoc network routing protocols turned out to be still vulnerable to attacks. This fact implies that the design of secure ad-hoc network routing protocols should be based on a systematic approach that minimizes the number of mistakes made in the design.

As an important step towards this goal, I propose a formal and automated method to verify the correctness of secure ad-hoc network routing protocols. I focus on the so called route forging attack, where the goal of the attackers is to achieve that an invalid route is accepted at the end of the route discovery session (by invalid route I mean an inexistent route in a given topology). My approach is based on a process calculus that I specifically designed for modeling the operation of secure ad-hoc network routing protocols, and an automated verification method based on logic based deductive and backward reachability analysis. I give the name sr-calculus for the proposed calculus, and the name sr-verif for the proposed automated verification method (and program), where the term sr refers to the words secure routing. I provide a systematic proof technique, called
BDSR, by combining the mathematical background of the sr-calculus and the backward deduction approach. The systematic nature of my method and its well-founded semantics ensure that one can have much more confidence in a security proof obtained by my method than in a “proof” based on informal arguments. In addition, compared to previous approaches that attempted to formalize the verification process of secure ad-hoc network routing protocols [18], [2], [3], [4], the novelty of my approach is that it can be fully automated. Furthermore, in contrast with [7] where the verification is made on a specific network topology, in my method it is performed on an arbitrary topology. Last but not least, with my proposed BDSR algorithm, the security of source routing protocols in presence of more than one attacker node can also be analyzed. My publications related to this topic are [Th05 , 2010], [Th06 , 2011], [Th07 , 2011], [Th08 , 2012].

2.2 Route forging attacks against secure routing protocols

Several “secure” routing protocols have been proposed in the recent past, however, later most of them are turned out to be vulnerable to various attacks. In this section, I introduce some known attacks against the SRP protocol [54], the Ariadne protocol [18], and the endairA protocol [3] that serve as the motivation of my work. Before going into details, I provide the formal definition of invalid route, which I refer to throughout this chapter.

Definition 1. (Invalid Route) Let $S_r$ be the set of edges in the route $r$, and let $S_T$ be the set of the subsets $S_{r_i}$, for all the routes $r_i$ in the topology $T$. We say that $r$ is an invalid route in $T$ if $S_r \notin S_T$.

2.2.1 The SRP protocol

Note that in my dissertation I consider the first version of SRP [54], and the setting when it uses shared keys between communicating pairs. SRP is a secure on-demand source routing protocol for ad-hoc networks, was first published in [54], and improved in [55]. The design of the protocol is inspired by the DSR protocol [37], however, DSR has no security mechanisms at all. Thus, SRP can be viewed as a secure variant of DSR. SRP tries to cope with attacks by using a cryptographic checksum in the routing control messages (route requests and route replies). This checksum is computed with the help of a key shared by the initiator and the target of the route discovery process.

In SRP, the initiator of the route discovery generates a route request message and broadcasts it to its neighbors. The integrity of this route request is protected by a Message Authentication Code (MAC) that is computed with a key shared by the initiator and the target of the discovery. Each intermediate node that receives the route request for the first time appends its identifier to the request and re-broadcasts it. The MAC in the request is not updated by the intermediate nodes, as by assumption, they do not necessarily share a key with the target. When the route request reaches the target of the route discovery, it contains the list of identifiers of the intermediate nodes that passed the request on. This list is considered as a route found between the initiator and the target.

The target verifies the MAC of the initiator in the request. If the verification is successful, then it generates a route reply and sends it back to the initiator via the reverse of the route obtained from the route request. The route reply contains the route obtained from the route request, and its integrity is protected by another MAC generated by the target with a key shared by the target and the initiator. Each intermediate node passes the route reply to the next node on the route (towards the initiator) without modifying it. When the initiator receives the reply it verifies the MAC of the target, and if this verification is successful, then it accepts the route returned in the reply. The message exchanges defined in the SRP protocol is illustrated in Figure 2.1.

The basic problem in (the first version of) SRP is that the intermediate nodes cannot check the MAC in the routing control messages. Hence, compromised intermediate nodes can manipulate control messages, such that the other intermediate nodes do not detect such manipulations.
2.2. Route forging attacks against secure routing protocols

Figure 2.1: The request and reply messages in (the first version of) SRP. rreq (rrep), S, D, ID, and SN
are the constant for specifying the message type, the IDs of the source and the target, the session ID, and
the sequence number, respectively. MACSD is the MAC computed over the tuple (rreq, S, D, ID, SN), by
the source using the key it shares with the target. MACTD is the MAC computed over the tuple (rrep, S,
D, [I1, I2]), by the target using the same shared key.

Furthermore, the accumulated node list in the route request is not protected by the MAC in the
request, hence it can be manipulated without the target detecting such manipulations.

In order to illustrate a known attack on (the first version of) SRP, let us consider the network
topology shown in Figure 2.2. Let us further assume that node N1 initiates a route discovery to
node N3. The attacker node A can manipulate the accumulated list of node identifiers in the route
request such that N3 receives the request with the list (N2, n, N4), where n is an arbitrary fake
identifier. This manipulation remains undetected, because the MAC computed by N1 does not
protect the accumulated node list in the route request, and intermediate nodes do not authenticate
the request. When the target N3 sends the route reply, A forwards it without modification to
N1 in the name of N2. As the route reply is not modified, the MAC of the target N3 verifies correctly
at N1, and thus, N1 accepts the route (N1, N2, n, N4, N3). However, this is a mistake, because
there is no link between N2 and n, and between n and N4.

Figure 2.2: An attack scenario against the first version of SRP. Note that this attack cannot be successful
in the improved version of SRP [55].

Note that the above attack has been found by manual analysis of the protocol. However, there
may be many similar attack scenarios (and indeed, there are), and manual analysis would be
inefficient to find all of them. The very purpose of my formal verification method to be introduced
in the upcoming sections of the dissertation is to make the analysis systematic and amenable for
automation such that it can efficiently find attacks against a protocol (within some limits of the
underlying model).

2.2.2 The Ariadne protocol

Ariadne has been proposed in [35] as a secure on-demand source routing protocol for ad hoc
networks. Ariadne comes in three different flavors corresponding to three different techniques for
data authentication. More specifically, authentication of routing messages in Ariadne can be based
on TESLA [56], on digital signatures, or on MACs. I discuss Ariadne with digital signatures.

There are two main differences between Ariadne and SRP. First, in Ariadne not only the
initiator and the target authenticate the protocol messages, but intermediate nodes too insert their
own digital signatures in route requests. Second, Ariadne uses per-hop hashing to prevent removal
of identifiers from the accumulated route in the route request. The initiator of the route discovery
generates a route request message and broadcasts it to its neighbors. The route discovery message
contains the identifiers of the initiator and the target, a randomly generated request identifier, and
2. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WIRELESS AD-HOC ROUTING PROTOCOLS

a MAC computed over these elements with a key shared by the initiator and the target. This MAC is hashed iteratively by each intermediate node together with its own identifier using a publicly known one-way hash function. The hash values computed in this way are called per-hop hash values. Each intermediate node that receives the request for the first time re-computes the per-hop hash value, appends its identifier to the list of identifiers accumulated in the request, and generates a digital signature on the updated request. Finally, the signature is appended to a signature list in the request, and the request is re-broadcast.

\[
\begin{align*}
S \rightarrow \ast: \text{Req} = (\text{rreq}, S, D, \text{ID}, h_{S}, [\text{ID}], [\text{sig}_S]) \\
I_1 \rightarrow \ast: \text{Req} = (\text{rreq}, S, D, \text{ID}, h_{S}, [\text{ID}], [\text{sig}_S]) \\
I_2 \rightarrow \ast: \text{Req} = (\text{rreq}, S, D, \text{ID}, h_{S}, [\text{ID}], [\text{sig}_S]) \\
D \rightarrow I_2: \text{Rep} = (\text{rrep}, S, D, [\text{ID}], [\text{sig}_S]) \\
D \rightarrow I_3: \text{Rep} = (\text{rrep}, S, D, [\text{ID}], [\text{sig}_S]) \\
I_1 \rightarrow S: \text{Rep} = (\text{rrep}, S, D, [\text{ID}], [\text{sig}_S]) \\
h_{S} = \text{MAC}_{\text{ID}}(\text{rreq}, S, D, \text{ID}), h_{S} = \text{hash}(\text{ID}, h_{S}), h_{S} = \text{hash}(\text{ID}, h_{S})
\end{align*}
\]

Figure 2.3: The request and reply messages in the Ariadne protocol. MAC$_{SD}$ is the MAC computed by the source using the shared key, $h_{I_1}$ is the hash computed on the concatenation of the ID $I_1$, and $h_{S}$, and $h_{I_2}$ is the hash computed on $I_2$ and $h_{I_1}$. Each signature $\text{sig}_I$ is computed over the whole packet portion before the signature.

When the target receives the request, it verifies the per-hop hash by re-computing the initiators MAC and the per-hop hash value of each intermediate node. Then it verifies all the digital signatures in the request. If all these verifications are successful, then the target generates a route reply and sends it back to the initiator via the reverse of the route obtained from the route request. The route reply contains the identifiers of the target and the initiator, the route and the list of digital signatures obtained from the request, and the digital signature of the target on all these elements. Each intermediate node passes the reply to the next node on the route (towards the initiator) without any modifications. When the initiator receives the reply, it verifies the digital signature of the target and the digital signatures of the intermediate nodes (for this it needs to reconstruct the requests that the intermediate nodes signed). If the verifications are successful, then it accepts the route returned in the reply.

Figure 2.4: A subtle attack against Ariadne. The figure on the left shows the communication during the route discovery, while the figure on the right illustrates that at the end of the route discovery phase, the source node accepts the route $S, V, W, A, D$, which is not valid because the link between $W$ and $A$ does not exist.

Despite these security mechanisms, it turns out that Ariadne is still vulnerable to route forging attack. I discuss here the attack scenario that has been published in [18] against Ariadne. Let us consider Figure 2.4, which illustrates part of a configuration where the discovered attack is possible. The attacker is denoted by $A$. Let us assume that $S$ sends a route request towards $D$. The request reaches $V$ that re-broadcasts it. Thus, $A$ receives the following route request message:

\[
\text{reqV} = (\text{rreq}, S, D, \text{ID}, h_{V}, [V], [\text{sig}_V])
\]

where $\text{ID}$ is the random request identifier, $h_{V}$ is the per-hop hash value generated by $V$, and $\text{sig}_V$ is the signature of $V$. 
2.2. Route forging attacks against secure routing protocols

After receiving $req_V$ the attacker waits until another copy of the same route request is received from $X$:

$$req_X = (rreq, S, D, ID, h_X, [V, W, X], [sig_V, sig_W, sig_X]).$$

From $req_X$, $A$ knows that $W$ is a neighbor of $V$. $A$ computes $h_A = H(A, H(W, h_V))$, where $h_V$ is obtained from $req_V$, and $H$ is the publicly known hash function used in the protocol. $A$ obtains the signatures $sig_V$, $sig_W$ from $req_X$. Then, $A$ generates and broadcasts the following request:

$$req_A = (rreq, S, D, ID, h_A, [V, W, A], [sig_V, sig_W, sig_A])$$

Later, $D$ generates the following route reply and sends it back towards $S$:

$$rep = (rreq, D, S, [V, W, A], [sig_V, sig_W, sig_A], sig_D).$$

When $A$ receives this route reply, it forwards it to $V$ in the name of $W$. Finally, $S$ will output the route $[S, V, W, A, D]$, which is a non-existent route.

2.2.3 The endairA protocol

The endairA protocol [3] was proposed after they found a security hole in the Ariadne protocol. The goal of endairA is to improve and revise the security solutions proposed in Ariadne, and to patch the security weaknesses can be found in it. The security mechanism of endairA uses less crypto functions, and per-hop signatures are used to protect the reply message which is the opposite of the solution in Ariadne.

In endairA, the initiator of the route discovery process generates a route request, which contains the identifiers of the initiator and the destination, and a randomly generated request identifier. Each intermediate node that receives the request for the first time appends its identifier to the route accumulated so far in the request, and re-broadcasts the request. When the request arrives to the destination, it generates a route reply. The route reply contains the identifiers of the initiator and the destination, the accumulated route obtained from the request, and a digital signature of the destination on these elements. The reply is sent back to the initiator on the reverse of the route found in the request. Each intermediate node that receives the reply verifies that its identifier is in the node list carried by the reply, and that the preceding identifier (or that of the initiator if there is no preceding identifier in the node list) and the following identifier (or that of the destination if there is no following identifier in the node list) belong to neighboring nodes. Each intermediate node also verifies that the digital signatures in the reply are valid and that they correspond to the following identifiers in the node list and to the destination. If these verifications fail, then the reply is dropped. Otherwise, it is signed by the intermediate node, and passed to the next node on the route (towards the initiator). When the initiator receives the route reply, it verifies if the first identifier in the route carried by the reply belongs to a neighbor. If so, then it verifies all the signatures in the reply. If all these verifications are successful, then the initiator accepts the route.

The operation and the messages of endairA are illustrated in Figure 2.5:

![Figure 2.5: The request and reply messages in the endairA protocol. Per-hop signatures are applied in the reply phase instead of the requests. Each signature $sig_i$ is computed over the whole packet portion before the signature.](image-url)
2. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WIRELESS AD-HOC ROUTING PROTOCOLS

Assuming one compromised node in the network, or several attacker nodes who cannot cooperate, the authors proved the security of endairA based on the simulation paradigm framework [3]. Burmeister et. al. [15] showed that when allowing compromised nodes to cooperate, endairA is vulnerable, and an invalid route can be accepted at the end of a route discovery session.

2.2.4 Summary

As we can see, in the attacks discussed in the previous subsections, the attacker node creates an incorrect routing state by modifying control messages during the route discovery phase so that the incorrect route is accepted as if it was correct. My emphasis is deliberately on modeling and verifying these kinds of subtle attacks. The motivation of my work lies in the fact that these kind of attacks are very tricky, thus, a formal and automatic verification methods are needed to discover and reasoning about them.

2.3 Related works

In this section, I will discuss the most important related works. In the literature, there are several formal languages, as well as automated model-checking tools for verifying different properties of systems and protocols, e.g., [27], [50], [58], [59] [57], [11], [64], [48]. These methods are not designed specifically for analyzing routing protocols, hence, their specification languages lack several syntax and semantics elements required for routing protocols (e.g., broadcast sending). Therefore, they cannot be used to analyze routing protocols, or only in a very circumstantial way, based on abstraction. In recent years, researchers focused on proposing specific methods for ad-hoc networks, e.g., [29], [30], [66], [7], [48], [18], [3], [71], [62]. However, the methods proposed in these related works have numerous drawbacks that I will discuss in the following two subsections.

2.3.1 Related formal analysis methods

In works [18], [3] the authors model the operation of the protocol participants by interactive and probabilistic Turing machines, where the interaction is realized via common tapes. This model enables us to be concerned with several feasible attacks. A so called security objective function is applied to the output of this model (i.e., the final state of the system) in order to decide whether the protocol functions correctly or not. Once the model is defined, the goal is to prove that for any adversary, the probability that the security objective function is not satisfied is negligible. The main drawback of this method is that the proof is not systematic and automated, and the framework is not well-suited for detecting attack scenarios once the proof fails. My goal is to improve these works by adding automated verification based on deductive model-checking.

In [27] the authors present the applied $\pi$-calculus that is a variant of the pure $\pi$-calculus [50]. The applied $\pi$-calculus is well-suited for modeling security protocols because it provides expressive syntax and semantics for reasoning on cryptographic primitives and operations. However, it lacks syntax and semantics for reasoning on broadcast communication, neighborhood, and communication range. Therefore, the applied $\pi$-calculus cannot be used directly for modeling routing protocols.

In order to give a formal and precise mathematical reasoning on the operation of routing protocols several process calculi have been proposed in the recent years. Among them the two works [29], [66] are closest to my work.

In the works [29], [30] the author proposes the process calculus, CMAN, for modeling mobile ad-hoc networks. The advantage of CMAN is that it includes syntax and semantics for modeling cryptographic primitives, neighborhood, broadcast communication. The main drawback of CMAN is that it does not provide syntax and semantics for modeling the accumulated knowledge of the attacker node, therefore, it cannot be directly used to model the attack scenario against SRP, Ariadne and endairA, that I showed in the Section 2.2 or the similar attacks presented in [18]. In these attacks the attacker node waits and collects information it intercepts during the route...
discovery, which it uses later to construct messages that contain incorrect route. In order to model these kind of attacks I propose the notion of the active substitution with range (detailed in Section 2.5.2) in my proposed calculus.

In [66] the authors propose the $\omega$-calculus. The main advantage of this calculus is that it has syntax and semantics for neighborhood, broadcast communication and mobility. The main drawback of this method is it does not provide syntax and semantics for modeling cryptographic primitives and the attacker's knowledge base. In contrast to the $\omega$-calculus my proposed calculus can model cryptographic primitives and attacker's accumulated knowledge can be explicitly modeled.

The advantage of these process calculi is that the operation of mobile ad-hoc networks and several properties such as loop-freedom and security properties can be precisely and systematically described with them, however, the drawback of them is that the proofs and reasoning are still performed manually by hand.

In [58], [59], the authors address the problem of formal analysis of secure neighbor discovery protocols (SND), and provide a novel formal verification method. Although the formalization and analysis of neighbor discovery (SND) is a bit different from the problem I address, because of the difference in the considered attacker scenarios and security goals, the authors provided a precise handling of the neighborhood and mobility, which are important in case of wireless ad-hoc networks.

### 2.3.2 Related automatic verification methods

Figure 2.6 shows the position of my contribution compared to previous works in the literature. In the figure, I classify the most important related works into three categories, each of which is represented by a circle. The circle on the left includes automatic model-checking tools, the uppermost circle contains the works that are concerned with formal analysis of ad-hoc and sensor networks, and finally, the formal methods proposed for reasoning about secured protocols can be found in the circle on the right.

SAL model checker, SPIN [57], and UPPAAL [11] are general purpose model-checking tools. The main drawback of them is that they lack semantics and syntax for modeling secure routing protocols, and for reasoning about attackers specific to ad-hoc networks. CSP [64], CPAL-ES [48], and ProVerif [13] are automatic verification tools developed for verifying security protocols, but they lack semantics and syntax for modeling routing protocols and ad-hoc networks. The tool in [66] is proposed for detecting loops in ad-hoc networks, however, it lacks semantics and syntax for modeling cryptographic primitives and operations, and does not consider attacker nodes.

A calculus for sensor networks [63], a calculus for ad-hoc networks [29], a work based on the simulation paradigm [18], and the $\omega$-calculus [66] are proposed for analyzing pure and secure routing protocols. However, the main drawback of these methods is that they are not automated.
The spi-calculus \[1\] and applied \(\pi\)-calculus \[27\] are proposed for modeling security protocols. They are not automated and cannot be used to model routing protocols.

To the best of my knowledge, my method is the first that supports all the three issues at the same time. The works that are the most closely related to my proposed sr-verif method are \([\text{Th05}, 2010]\) and \([13]\). The main novelty of sr-verif compared with the related methods is that the verification is performed on arbitrary network topologies, besides avoiding the exponential state explosion during the verification. Similar to the proof method in the \(sr\)-calculus, sr-verif also based on the BDSR algorithm. sr-verif was inspired by the verification method used in the broadly used Proverif automatic verification tool proposed for verifying security protocols \([13]\). However, as opposed to \([13]\), sr-verif is designed for verifying routing protocols, it includes numerous novelties such as the modeling of broadcast communications, neighborhood, transmission range, and it uses an attacker model specific to wireless ad hoc networks.

Based on their characteristics, my proposed sr-verif and ProVerif (along with other logic based verifiers, such as CSP, CPAL-ES) can be seen as fully automated theorem provers (although the authors of these verifiers do not make an explicit statement about the type of their tools). Model-checkers (SAL, SPIN, UPPAL) are fully automated. The system to be analysed is defined in finite state machine (Kripke-structure) and the goal is defined by temporal logic (e.g. LTL, CTL). The verification is based on the checking if the language of the model is the subset of the property. Theorem provers are fully logic based, however, they are not always fully automated, and requires human interaction. Main function of theorem provers is to aid researchers to prove some theorems.

### 2.4 Assumptions on routing protocols and the attacker model

I assume that routing protocols are composed of a request and a reply phase. In the request phase the route request for a session is initiated by the source, while in the reply phase the route reply is sent by the destination. I focus on verifying on-demand source routing protocols in which the information about the route is included in request and reply messages in form of an ID list.

In the following, some typical properties of source routing protocols are given. Every honest node checks ID duplication in ID lists. When an intermediate node receives a request or a reply message, it checks if its ID is in the ID list, and the next and previous IDs belong to its neighbors. If this is not the case, then the message is dropped. The source checks the first whilst the destination checks the last ID in the received ID list. Furthermore, I assume that the source routing protocol is specified such that each honest intermediate node appends its own ID into the ID list in the request before forwarding it, and a reply message contains information (implicit of explicit) about the addressee. Of course, this assumption is not necessarily valid to the messages sent by the attackers, because they can modify the content of the messages. These assumptions are valid to all the representative on-demand source routing protocols DSR, SRP, Ariadne, endairA, where in the request message the last node ID in the list belongs to the sender node, while both the addressee and the sender are encoded in a reply message.

Within a session, every intermediate node considers only the request it receives for the first time, further requests with the same header are dropped. The destination can accept several requests, and the source can accept several replies. For increasing the efficiency of the verification, I assume that the attackers cannot obtain the secret keys of the honest nodes. The rationale behind this assumption is that in most cases the route forging attacks can be performed without knowing the secret keys.

I assume one or several attacker nodes which are compromised nodes, meaning that they can perform computations like honest nodes, and possess information that honest nodes can have according to the protocol. But unlike the honest nodes, attacker nodes can either decide to follow the protocol or not. In the latter case attacker nodes can modify messages, and when it intercepts a request it can remain idle and does nothing, or it can forward messages unchanged. Attacker nodes can cooperate with each other, and they can run parallel sessions of the protocol at the same time.

I do not assume direct communication channels between the attacker nodes. The reason is that
the attackers with common links (i.e., neighbor attacker nodes) can be merged into one attacker
node which inherits the links (neighbors) of all the attackers. Hence, any attack scenario that can
be found in case of many attackers who are neighbors of each other, is also valid after merging all
the attacker nodes into one “super” attacker node.

An important observation is that in order to perform an attack, the attackers cannot stay idle
after intercepting a reply in the reply phase. Let us assume the opposite, i.e. the attackers stay
idle after intercepting the reply and an invalid route is accepted by the source at the end. By
assumption, we have that the invalid route reply gets back to the source without passing through
the attacker. However, due to the fact that every intermediate node checks its neighbors, the
invalid route reply cannot reach the source via only honest nodes.

2.5 The proposed sr-calculus

In this section, I define the proposed calculus: Its type system and formal syntax, as well as
its operational semantics. The advantage of the sr-calculus is that its expressiveness allows for
modeling broadcast communication, neighborhood, and transmission range like CMAN and the
ω-calculus, and cryptographic primitives like the applied π-calculus, however, compared to them
it includes novelties such as the definition of active substitution with range that enables us to
model the attacker’s knowledge base and attacks in the context of wireless ad-hoc networks. More
precisely, the sr-calculus can be seen as the combination of the three calculi with some modifications
and extensions. I also provide a novel definition of bisimilarity for reasoning about the security of
ad-hoc network routing protocols against the class of route forging attacks.

2.5.1 Type system of the sr-calculus

I provide a basic type system for the proposed calculus. The main purpose of the type system
is to reduce the number of the possible cases to be examined during the formal security proofs.
Based on the type system we are capable of capturing errors such as arity mismatch and erro-
neous binding/substitution of terms. I adopt the type system proposed for the applied π-calculus,
discussed in the chapter 4 of [14], which have been shown to be sound and complete. This type
system includes a syntax and a semantics part, which discuss the declaration of the types and the
rules for typing, for example, the type preserved property of transitions.

The type system catches the errors such as arity errors and the binding of terms with mismatch
types. The type system does not include recursive types, hence, processes such as τ⟨c⟩.P has
undefined type.

Definition 2. Type assignment is an assignment v : T of a type T to v (or u) that can be a name,
a constant, a node ID or a variable.

The set of types is divided into the sets of term types and process/behavior types. Within the
term types, I distinguish among channel types, broadcast types, name types, variable types, constant
types, and node ID types. Within the node ID types, I also distinguish between IDs of honest nodes
and IDs of attacker nodes.

Given a term type Ti, channel and broadcast channel types are constructed by the unary type
constructors ch(Ti) and bch(Ti), which are the types that is allowed to carry data with term type.

The types for the sr-calculus are generated by the grammar:

\[
\begin{align*}
S, T ::&= T_i | T_{proc} & \text{(Types)} \\
T_i ::&= T_{ch} | T_{br} | T_{str} & \text{(Term Types)} \\
T_{str} ::&= T_{name} | T_{var} | T_f | T_{const} & \text{(String Types)} \\
T_{name} ::&= t_{n1} | \ldots | t_{nn} & \text{(Name Types)} \\
T_{var} ::&= t_{v1} | \ldots | t_{vn} & \text{(Variable Types)} \\
T_f ::&= f(T_{str}^{1}, \ldots, T_{str}^{n}) & \text{(Function Types)} \\
T_{const} ::&= T_{req/rep} | t_{const} & \text{(Constant Types)}
\end{align*}
\]
Routing Protocols


where $tn$, $tv$, $tl$ and $tp$ are name, variable, node ID, and process types, respectively. The abbreviation of $x_1 : T_1, \ldots, x_n : T_n$ is defined by $\bar{x} : \bar{T}$. Of course, if a term $t$ has a string type $T_{str}$ then it also has a term type $T_t$, and if $t$ has been assigned to one of the type $T_{name}$, $T_{var}$, $T_f$, $T_{const}$ then it implicitly has a type $T_{str}$. The reverse phase is not always true, hence, to avoid type conflict the most narrow type should be assigned in the declaration. Note that within the set of term type the channel types are distinguished from the remaining string types because to reasoning about routing protocols we do not need to send channels, or need not to define a function that includes channel arguments. Within the constant type I define $T_{req/rep}$ as the type for the special constants $rreq$ and $rrep$ which are the parts of the routing messages.

Within a function types I distinguish types of each crypto function, such as, digital signature type, $T_{sig}$, one-way hash type, $T_{hash}$, MAC function type, $T_{mac}$. I also define types of secret key, $T_{skkey}$, public key $T_{pkkey}$, and symmetric shared key $T_{shkey}$. In the dissertation, only these three crypto functions are used, but of course any function types can be defined in the similar way. With these types we can ease the security verification, and reducing the number of possibilities.

$$
\begin{align*}
T_{skkey} & ::= sk(T_{id}) & \text{(Secret Key Types)} \\
T_{pkkey} & ::= pk(T_{id}) & \text{(Public Key Types)} \\
T_{shkey} & ::= k(T_{id}, T_{id}) & \text{(Shared Key Types)} \\
T_{sig} & ::= sig(T_{str}, T_{skkey}) & \text{(Digital Signature Types)} \\
T_{hash} & ::= hash(T_{str}) & \text{(One-Way Hash Types)} \\
T_{mac} & ::= mac(T_{str}, T_{key}) & \text{(MAC Types)}
\end{align*}
$$

The syntax, the reduction rules and the transition rules for the typed applied $\pi$-calculus remains unchanged from the one for the untyped applied $\pi$-calculus.

2.5.2 Formal Syntax of the $st$-calculus

I assume an infinite set of names $N$ and variables $V$, where $N \cap V = \emptyset$. Further, I define a set of node identifiers (node ID) $L$, where $N \cap L = \emptyset$. Each node identifier uniquely identifies a node. Below the definition of term, denoted by $t$, is given:

$$
t ::= rreq \mid rrep \mid \text{Accept} \mid \text{undef} \mid \text{true} \mid c \mid n \mid l_1 \mid l_{src}, l_{dest}, l_a, \mid x_{index} \mid y_{this}, y_{prov}, y_{nxt}, y_{honprov}, y_{honnxt}, y_{idx} \mid t_a \mid f(t_1, \ldots, t_k).
$$

Terms take their values from a set of data of different types, namely

- $rreq$ and $rrep$ are unique constants that represent the $rreq$ and $rrep$ tags in route request and reply messages;
- $\text{Accept}$ is a special constant. The source node outputs $\text{Accept}$ when it receives the reply message and all the verifications it makes on it are successful. Namely, the ID list included in the reply is accepted.
- $\text{undef}$ and $\text{true}$ are special constants that I use during the analysis of the protocols. More details will be given in Section 2.6.
- $c$ models communication channels for unicast communication.
2.5. The proposed sr-calculus

- \( n \) is a name and models some elemental data;
- \( l_i, i \in \{1, \ldots, k\} \) represents an ID of the honest intermediate node;
  \( l_{\text{src}}, l_{\text{dest}}, l_{\text{att}}, j \in \{1, \ldots, h\} \) are the IDs of the source, destination, and the attackers, respectively.
- \( x_{\text{index}} \) is a variable that models any term, that is, it has variable type;
  \( y_{\text{nid}}, y_{\text{prv}}, y_{\text{nxt}} \) represent variables of type node ID, thus, both the IDs of the attacker and the honest node can be bounded to them.
  \( y_{\text{this}}, y_{\text{honprv}} \) and \( y_{\text{honnxt}} \) define the variables of type honest node’s ID, hence, only \( l_i \) can be bounded.
- \( l_{a} \) be the variable of type \( T_{\text{attid}} \), such that only \( l_a \) can be bounded.
- \( f(t_1, \ldots, t_k) \) is a function with arity \( k \) defined on terms. Function is used to model cryptographic primitives, route request and reply messages. For instance, digital signature is modeled by the function \( \text{sign}(t_1, t_2) \), where \( t_1 \) models the message to be signed and \( t_2 \) models the secret key, and \( f \) is \( \text{sign} \).

Note that this definition of term differs from CMAN, the \( \omega \)-calculus, and the applied \( \pi \)-calculus in that it includes the constants \( \text{rep}, \text{req}, \text{Accept} \), the node IDs and corresponding variables for analyzing routing forging attacks.

The internal operation of nodes is modeled by \textit{processes}. Processes can be specified with the following syntax, and inductive definition:

\[
P, Q, R ::= \text{processes} \quad \tau(t).P \mid c(x).P \mid \langle t \rangle.P \mid (\langle t \rangle).P \mid (P|Q) \mid \nu n.P \mid !P \mid [t_1 = t_2]P \mid [l \in \sigma]P \mid \text{nil} \mid \text{let} (x = t) \text{ in } P;
\]

- The process \( \tau(t).P \) represents the unicast sending of message \( t \) on channel \( c \), followed by the execution of \( P \).
- The process \( c(x).P \) receives some message on channel \( c \) and binds it to every variable \( x \) in process \( P \).
- The process \( \langle t \rangle.P \) represents the broadcast of message \( t \), continued with the process \( P \).
- The process \( (\langle t \rangle).P \) represents the receive of some broadcast message which will be bounded to each occurrence of \( x \) in \( P \). Compared to the unicast case the two broadcast processes do not contain any certain channel, which intends to model that there is not a specified target.
- Process \( P|Q \) is the \textit{parallel composition} of processes \( P \) and \( Q \) and behaves as processes \( P \) and \( Q \) running in parallel: each may interact with the other, or with the outside world, independently from the other.
- A restriction \( \nu n.P \) is a process that creates a new, private name \( n \), and then behaves as \( P \).
- Process \( !P \) represents the \textit{infinite replication} of process \( P \). This process is equivalent to the parallel composition of infinite number of \( P \) instances, \( P \mid P \mid \ldots \).
- Processes \( [t_1 = t_2]P \) and \( [l \in \sigma]P \) mean that if \( t_1 = t_2 \) and \( l \in \sigma \), respectively, then process \( P \) is ”activated”, else it gets stuck and stays idle.
- The \textit{nil} process \textit{nil} does nothing.
- Process \textit{let} \( t = x \text{ in } P \) represents the binding of term \( t \) to every variable \( x \) that occurs in process \( P \).
Note that this definition of processes differs from the $\omega$-calculus because constructor and destructor applications, used to model cryptographic primitives, are also included here. Compared to CMAN sr-calculus includes $[l \in \sigma]P$ and possibility for unicast. Finally, differs from the applied $\pi$-calculus, here broadcast send and receive actions are included.

Nodes are defined as $[P]_l^i$ which represents a node that has the identifier $l$ and behaves as $P$, and its transmission range covers the nodes whose identifiers are in the set $\sigma$. Two nodes are neighbors if they are in each other’s range. Note that $\sigma$ can be empty and a node is $[P]_l^i$, which means that the node has no connections.

A network, denoted as $N$, can be an empty network with no nodes: $0_N$; a singleton network with one node: $[P]_l^i$; the parallel composition of nodes: $[P_1]_{l_1}^{\sigma_1} | [P_2]_{l_2}^{\sigma_2}$, where $\sigma_1$ and $\sigma_2$ may include $l_2$ and $l_1$ respectively; and the composition of networks: $N_1 \parallel N_2$.

Finally, in order to model attackers that improve their knowledge by accumulating information they hear from their neighbors, I extend the syntax of networks with the active substitution with range: An extended network $E$ is defined as follows:

- An extended network can be a plain network $N$ that I already discussed above.
- An extended network can be a parallel composition of two extended networks: $E_i | E_j$.
- An extended network is equipped with the active substitution with range $\{t/x\}^\sigma$. This says that the substitution $\{t/x\}$ binds term $t$ to every variable $x$ that occurs in any node $[P]_l^i$, that is in parallel composition with $\{t/x\}^\sigma$, and $l_i \in \sigma$. Intuitively, $\sigma$ is the range of the substitution $\{t/x\}$. Again, note that the notion of active substitution with range is novel compared with CMAN, the $\omega$-calculus and the applied $\pi$-calculus.

This part is new compared to CMAN, and the $\omega$-calculus in that it enables us to model the knowledge base of the attacker node. The knowledge of the attacker can improve after a series of communication steps. Also, this part is novel compared to the applied $\pi$-calculus in that active substitution has range for modeling neighborhood.

### 2.5.3 Operational Semantics of the sr-calculus

The operational semantics of sr-calculus is defined by the internal reduction rules, structural equivalences and the labeled transition system (LTS) that composed of labeled transition rules.

The internal reduction rules model the internal computation of nodes such as comparing two terms, while the labeled transition rules model the communication of nodes such as broadcast send and receive.

#### (Internal reduction rules)

- **(Red. Let)** $[\text{let } x = t \text{ in } P]_l^i \rightarrow [P[t/x]]_l^i$
- **(Red. If1)** $[[t = l]P]_l^i \rightarrow [P]_l^i$
- **(Red. If2)** $[[t = s]P]_l^i \rightarrow \text{nil}_l^i$ (if $t \neq s$)
- **(Red. In1)** $[[l \in \sigma]P]_l^i \rightarrow [P]_l^i$ (if $l \in \sigma$)
- **(Red. In2)** $[[l \in \sigma]P]_l^i \rightarrow \text{nil}_l^i$ (if $l$ is not in $\sigma$)

The internal reduction relation is denoted by $\rightarrow$. Rule (Red. Let) models the binding of term $t$ to variable $x$ in process $P$. The rules (Red. If1) and (Red. If2) model the equality check of two terms. If the two terms are equal then the internal operation is followed by $P$, else it gets stuck; Finally, the rules (Red. In1) and (Red. In2) check if the node ID $l$ is in the set $\sigma$.

In addition, internal reduction rules can be used to model mobility of nodes by the following rules:

#### (Reduction rules for mobility)

- **(Red. Connect)** $[P]_{l_1}^{\sigma_1 \cup l_2} | [Q]_{l_2}^{\sigma_2} \rightarrow_{\{l_1 \circ l_2\}} [P]_{l_1}^{\sigma_1} | Q]_{l_2}^{\sigma_2}$, where $l_2$ is not in $\sigma_1$.
- **(Red. Disconnect)** $[P]_{l_1}^{\sigma_1 \cup l_2} | [Q]_{l_2}^{\sigma_2} \rightarrow_{\{l_1 \circ l_2\}} [P]_{l_1}^{\sigma_1} | Q]_{l_2}^{\sigma_2}$, where $l_2$ is not in $\sigma_1$. 


The rule (Red. Connect) model the scenario in which node $l_2$ gets into the transmission range of the node $l_1$. This reduction relation is denoted by $\rightarrow_{\{l_1 \bullet l_2\}}$. Its counterpart is the rule (Red. Disconnect) says that node $l_2$ gets out of the transmission range of the node $l_1$. This reduction relation is denoted by $\rightarrow_{\{l_1 \circ l_2\}}$.

In the following, the most important labeled transitions are presented. Labeled transitions are used to model the communication of nodes, and play important roles in proofs. Labeled transitions are denoted by a labeled relation that is an arrow with a label: $\rightarrow_{\alpha}$.

**(Labeled transition rules)**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Br-snd) $\frac{(t).P</td>
<td>_{l_1}^{\sigma} \stackrel{\nu x.(x)_l^{\sigma}}{\rightarrow} (t/x)^{\sigma}</td>
</tr>
<tr>
<td>(Br-rcv) $\frac{(t/x)^{\sigma}</td>
<td>(t)_l^{\sigma}</td>
</tr>
</tbody>
</table>

The first rule means that node $l$ broadcasts $t$, so that $t$ is now available for all nodes in its transmission range $\sigma$. This is modeled by the active substitution with range $\{t/x\}^{\sigma}$ and $\nu x$, which restricts the substitution to the nodes that are within the range $\sigma$. The second rule means that if the listening node $l$ is within the range $\sigma$ then it receives the broadcast $t$.

Finally, next I introduce the most important structural equivalence rules on extended networks. The structural equivalence relation is denoted by $\equiv$. Using structural equivalence rules we can tell if two networks are identical up to structure.


<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Struct. E-Par1)</td>
<td>$E[0_N \equiv E$</td>
</tr>
<tr>
<td>(Struct. E-Par2)</td>
<td>$E_1</td>
</tr>
<tr>
<td>(Struct. E-Par3)</td>
<td>$E_1(E_2</td>
</tr>
<tr>
<td>(Struct. E-Try)</td>
<td>${t/x}^{\sigma}E \equiv {t/x}^{\sigma}E[t/x]^{\sigma}$</td>
</tr>
<tr>
<td>(Struct. E-Rewrite)</td>
<td>${t_1/x}^{\sigma} \equiv {t_2/x}^{\sigma}$ (if $t_1 = t_2$)</td>
</tr>
</tbody>
</table>

The first rule says that the composition with an empty network does not change anything. The next two rules concern the commutative and associative properties. Rule (Struct. E-Rewrite) says that two active substitutions with the same range $\sigma$ and terms are structurally equivalent. Rule (Struct. E-Try) tries to apply the active substitution with range $\{t/x\}^{\sigma}$ to the extended network $E$: For example, let $E$ be

$$E = \{t_1/x_1\}^{\sigma_1} \ldots \{t_k/x_k\}^{\sigma_k} \mid |Q_i|_{l_i}^{\sigma_i} \ldots |Q_j|_{l_j}^{\sigma_j}$$

Then $E[t/x]^{\sigma}$ is

$$\{t_1/x_1\}^{\sigma_1} \ldots \{t_k/x_k\}^{\sigma_k} \mid |Q_i|_{l_i}^{\sigma_i} \{t/x\}^{\sigma} \ldots \{t_j|_{l_j}^{\sigma_j} \{t/x\}^{\sigma}$$

Intuitively, this means that the substitution is tried on every plain network. However, this substitution takes place at $|Q_i|_{l_i}^{\sigma_i}$ only if $l_i \in \sigma$. This is formally defined by a new rule (E-Subst) and the relation $\equiv_{l_i \in \sigma}$ below:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Struct. E-Subst)</td>
<td>${t/x}^{\sigma}</td>
</tr>
</tbody>
</table>

Note that these labeled transition rules and structural equivalence rules are novel compared to the related works I mentioned above due to the application of the active substitution with range in each rule. In addition, the rules (Red. In1) and (Red. In2) are also new compared to the related calculi.

### 2.5.4 Attacker knowledge base, static equivalence, labeled bisimilarity

In this subsection, I give the definition of labeled bisimilarity that defines the equivalence between two wireless ad-hoc networks, meaning that they cannot be distinguished by an observer which can eavesdrop on communications. I also discuss the definition of static equivalent borrowed from
2. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WIRELESS AD-HOC ROUTING PROTOCOLS

the applied \( \pi \)-calculus, and adjust it to my problem, that is the static equivalent of two networks. Finally, I also provide the notion of frame that is used to capture the knowledge based of the observer, and the attacker nodes.

Let \( \mathcal{L}(N) \) be the set of identifier \( l_i \)'s in the network \( N \), again recall that each \( l_i \) is a unique name in the network, and identify each node of the network. Then let connect\(_j\)(\( \mathcal{L}(N) \)) \( \in \mathcal{C}(N) \) be a set of all links in the \( j \)-th topology of the network \( N \). \( \mathcal{C}(N) \) is the set of all possible topologies of \( N \).

Recall that an extended network is composed of active substitution with range and plain networks as follows:

\[
E = \nu \tilde{n}. (\{t_1 / x_1\}^{\sigma_1} | \{t_2 / x_2\}^{\sigma_2} | \ldots | \{t_m / x_m\}^{\sigma_m} | N_1 | \ldots | N_r)
\]

The output of the extended network \( E \) is defined by a frame \( \varphi \), which is composed of name restrictions and a parallel composition of all active substitutions:

\[
\varphi = \nu \tilde{n}. (\{t_1 / x_1\} | \{t_2 / x_2\} | \ldots | \{t_m / x_m\})
\]

Note that in \( \varphi \) the ranges \( \sigma_1, \ldots, \sigma_n \) are removed from active substitutions with range.

Intuitively, the frame represents the output of the network. Note that while the attacker node knows only the messages sent by its neighbors the wireless environment knows all of the sent messages. By observing everything, the wireless environment can distinguish the operation of two networks. When an attack (against a routing protocol) is executed successfully on a specific topology, the wireless environment will be aware of it, because it can distinguish the correct operation (i.e., follows the protocol) from the incorrect operations.

Taking into account that in our adversary model the attacker is weaker than the Dolev-Yao attacker in the sense that he cannot eavesdrop all the messages sent in the network but only the messages sent by its neighbors the wireless environment knows all of the sent messages.

By adapting the notion of frame [27], the accumulated knowledge base of the attacker is defined as the frame with the identifier \( l_a \) as parameter: \( \varphi(l_a) \). The frame \( \varphi(l_a) \) can be seen as the "subset" of the frame \( \varphi \), because it contains only such active substitution(s) \( \{t_i / x_i\}^{\sigma_i} \), where \( l_a \in \sigma_i \). That is,

\[
\varphi(l_a) = \nu \tilde{n}. (\{t_1 / x_1\} | \{t_2 / x_2\} | \ldots | \{t_k / x_k\})
\]

where \( l_a \in \sigma_1, l_a \in \sigma_j, \ldots, l_a \in \sigma_k \), and \( \{i,j, \ldots, k\} \subseteq \{1, 2, \ldots, n\} \).

**Definition 3.** Two terms \( t_1 \) and \( t_2 \) are equal in a frame \( \varphi \), and write \( [t_1 = t_2] \varphi \), if and only if \( \varphi \equiv \nu \tilde{n}. \omega \), \( t_1 \omega = t_2 \omega \), and \( \{\tilde{n}\} \cap (fn(t_1) \cup fn(t_2)) = \emptyset \) for some names \( \tilde{n} \) and substitution \( \omega \).

**Definition 4.** Two closed frames \( \varphi \) and \( \psi \) are statically equivalent, and write \( \varphi \approx_s \psi \), when \( \text{dom}(\varphi) = \text{dom}(\psi) \) and when, for all terms \( t_1 \) and \( t_2 \), we have \( [t_1 = t_2] \varphi \) if and only if \( [t_1 = t_2] \psi \). We say that two closed extended networks are statically equivalent, and write \( E_1 \approx_s E_2 \), when their frames are statically equivalent.

We say that two extended networks \( E_1 \) and \( E_2 \) are statically equivalent, denoted as \( E_1 \approx_s E_2 \), if their frames are statically equivalent. Two frames \( \varphi_1 \) and \( \varphi_2 \) are statically equivalent if they includes the same number of active substitutions and same domain; and any two terms that are equal in \( \varphi_1 \) are equal in \( \varphi_2 \) as well. Intuitively, this means that the outputs of the two networks cannot be distinguished. The advantage of the static equivalence is that it does not depend on the arbitrary environment of processes. Instead, in order to check the validity of the equivalence it is enough to verify the frames that captures the outputs of a network so far.

Finally, I define the labeled bisimilarity in the context of wireless ad-hoc network, which is novel compared to CMAN, the \( \omega \)-calculus, and the applied \( \pi \)-calculus. The advantage of the labeled bisimilarity is that it does not depend on an arbitrary context but only on the frames which is known after each transition step. Note that I am considering only the reasoning about the attacks discussed in Section 2.2, hence, I assume that the topology remains unchanged during attack. In the next definition, \( E_1 \) is specified that it consists of one plain network \( N_1 \), and \( E_2 \) consists of one plain network \( N_2 \).
Definition 5. Labeled bisimilarity for a given network topology \( (\approx^N) \) is the largest symmetric relation \( \mathcal{R} \) on closed extended networks such that \( E_1 \mathcal{R} E_2 \) implies: \( \mathcal{L}(N_1) = \mathcal{L}(N_2) \) and \( \text{connect}_1(\mathcal{L}(N_1)) = \text{connect}_2(\mathcal{L}(N_2)) \), and

1. \( E_1 \approx_1 E_2 \);
2. if \( E_1 \to E'_1 \) and \( E_2 \to^* E'_2 \) and \( E'_1 \mathcal{R} E'_2 \) for some \( E'_2 \); (This is the induction based on internal reductions);
3. if \( E_1 \overset{\alpha}{\to} E'_1 \) and \( \text{fn}(\alpha) \subseteq \text{dom}(E_1) \) and \( \text{bn}(\alpha) \cap \text{fn}(E_2) = \emptyset \); then \( E_2 \to^* E'_2 \) and \( E'_1 \mathcal{R} E'_2 \) for some \( E'_2 \). (This is the induction based on labeled relations). Here \( \alpha \) can be a broadcast, an unicast, or a receive action.

Intuitively, this means that the outputs of the two networks of same topology cannot be distinguished by the wireless environment during their operation. In particular, the first point means that at first \( E_1 \) and \( E_2 \) are statically equivalent; the second point says that \( E_1 \) and \( E_2 \) remains statically equivalent after internal reduction steps. Finally, the third point says that if the node \( l \) in \( E_1 \) outputs (inputs) something then the node \( l \) in \( E_2 \) outputs (inputs) the same thing, and the "states" \( E'_1 \) and \( E'_2 \) they reach after that remain statically equivalent. Here, \( \to^* \) models the sequential execution of some internal reductions, or more formally, a transitive and reflexive closure of \( \to \).

Definition 6. Given \( E_1 \) and \( E_2 \) such that \( \mathcal{L}(N_1) = \mathcal{L}(N_2) \), we say that \( E_1 \) and \( E_2 \) are labeled bisimilar if they are labeled bisimilar for every topology. That is, \( \forall \text{connect}_1 \in \mathcal{C}(N_1), \forall \text{connect}_2 \in \mathcal{C}(N_2) \), such that \( \text{connect}_1(\mathcal{L}(N_1)) = \text{connect}_2(\mathcal{L}(N_2)) \) : \( E_1 \approx^N E_2 \).

In order to verify the security of a given source routing protocol, \( \text{routeprot} \), based on the labeled bisimilarity, I define two \( sr \)-calculus specifications for \( \text{routeprot} \), namely, the real protocol definition and the ideal definition. The real specification of \( \text{routeprot} \) is an extended network, denoted by \( E_{\text{real}} \text{routeprot} \), which follows exactly the (informal) definition of the \( \text{routeprot} \) routing protocol. The ideal specification of \( \text{routeprot} \) is an extended network, denoted by \( E_{\text{ideal}} \text{routeprot} \), which is defined in the same way as \( E_{\text{real}} \text{routeprot} \), except for the specification of the source node. The only difference between \( E_{\text{real}} \text{routeprot} \) and \( E_{\text{ideal}} \text{routeprot} \) is that in \( E_{\text{ideal}} \text{routeprot} \), the source node is able to check the validity of the returned route, independently of what the attacker(s) may do. With this setting, the Definition 6 can be rephrased as follows:

Definition 7. Let \( E_{\text{real}} \text{routeprot} \) and \( E_{\text{ideal}} \text{routeprot} \) be the real and the ideal specifications of a given source routing protocol, \( \text{routeprot} \). The routing protocol, \( \text{routeprot} \), is said to be secure if for every network topology \( N \) and the same attacker nodes, we have: \( E_{\text{real}} \text{routeprot} \approx^N E_{\text{ideal}} \text{routeprot} \).

Note that in my dissertation, I mainly focus on the class of route forging attacks, and the labeled bisimilarity assumes a restrictive correctness specification. There are several other specifications for correctness of secure route discovery, given in the paper [55]. In [55], the authors examine different correctness definitions such as loop-freedom and freshness, which also taking into account the dynamics of wireless networks.

The soundness of the \( sr \)-calculus is based on the soundness of the applied \( \pi \)-calculus, the omega-calculus and CMAN. I proved that the modification I made on the existing sound syntax and operational semantics and definitions of the three calculi preserves the soundness property. Due to page limitation I refer the reader to the report [Th08 , 2012] for the proof.

Attacker’s ability and knowledge

The knowledge of the attacker is composed of initial knowledge (denoted by \( K_{\text{init}} \)) and gained knowledge (denoted by \( K_{\text{gain}} \)). Typically, \( K_{\text{init}} \) often contains the node IDs of the neighborhood of the attacker, and keys that the attacker shared with the honest nodes. The initial knowledge of the attacker is modeled by the frame \( \varphi(l_a) \), which is initialized with the substitutions that binds the initial knowledge of the attacker to variables. Thereafter, during the protocol run, the
frame \( \varphi(l_a) \) is periodically extended with new information obtained by the attacker (i.e., \( \mathcal{K}_{gain} \)). Whenever the attacker computes a message for his purpose it can use its entire set of knowledge and computation ability.

The computation ability of the attacker is an unchanged set of constructor functions such as computing encryption, hash, and digital signature, compose a message tuple etc. The computation ability of the attacker node, denoted by \( \mathcal{K}_{att} \), is the set of functions such as \( encrypt(t,k) \), \( hash(t) \), \( mac(t,k) \), \( sign(t,k) \), etc. To capture the attacker’s ability for message verification, set \( \mathcal{K}_{att} \) also contains equations from the equational theory like in the applied \( \pi \)-calculus, such as \( dec(enc(x, y), y) = x \) and \( checksign(sign(x, y), pk(y)) = x \). In the processes of the attacker, the parameters of these functions and equations can only those that appear in \( \varphi(l_a) \).

### 2.5.5 Examples

In the following, using three example scenarios I demonstrate the application of active substitution with range, and the defined labeled transition system.

#### Example for broadcasting and message loss

The first example network illustrated in the Figure 2.7 includes three nodes. In this simple network, node \( P \) is assigned the identifier \( l_1 \), nodes \( Q \) and \( R \) have the identifiers \( l_2 \) and \( l_3 \), respectively. Node \( P \) and node \( Q \) are neighbors, but node \( P \) and \( R \) are not. Thus, when \( P \) broadcasts the message \( t \), only \( Q \) receives it. The following labeled transitions model the procedure in which \( P \) broadcasts \( t \), and only \( Q \) receives this.

\[
\begin{align*}
Q \xrightarrow{t} P & \xrightarrow{t} R
\end{align*}
\]

**Figure 2.7:** An example network with three nodes, and a link between \( P \) and \( Q \).

\[
\left( \left[ (t).P_1 \right]_{t_1}^{l_2}, \left[ (y).Q_1 \right]_{t_2}^{l_1}, \left[ (z).R_1 \right]_{t_3} \right) \xrightarrow{\nu x.(x).\left[ (t) \right]_{l_2}^{l_2}}
\]

\[
\left( \left[ t/x \right]_{t_1}^{l_2}, \left[ y.Q_1 \right]_{t_2}^{l_1}, \left[ (z).R_1 \right]_{t_3} \right) \xrightarrow{(t_2)^{(l_1,l_2)}}
\]

\[
\left( \left[ t/x \right]_{t_1}^{l_2}, \left[ y.Q_1 \right]_{t_2}^{l_1}, \left[ (z).R_1 \right]_{t_3} \right) \xrightarrow{(t_2)^{(l_1,l_2)}}
\]

\[
\left[ (t).P_1 \right]_{t_1}^{l_2}, \left[ (y).Q_1 \right]_{t_2}^{l_1}, \left[ (z).R_1 \right]_{t_3}
\]

represent the nodes \( P \), \( Q \) and \( R \), respectively. This example includes only one broadcast step, and there is no replication. First, the rule (Br-snd) is applied, which is followed by the rule (Br-rcv), which says that \( Q \) receives \( t \), because \( l_2 \in \{l_1, l_2\} \).

#### Example for multiple broadcast sends and receives

The next example is a little more complicated, which also includes replication and multiple broadcasts. The network can be seen in the Figure 2.8. In this example, both \( P \) and \( Q \) are the neighbors of \( R \). First, \( P \) broadcasts \( t_1 \) and then \( Q \) broadcasts \( t_2 \). Node \( R \) is under replication which means that it repeatedly listens for messages.

\[
\begin{align*}
P \xrightarrow{t_1} R \xrightarrow{t_2} Q
\end{align*}
\]

**Figure 2.8:** Another example network

\[
\left( \left[ (t_1).P \right]_{t_1}^{l_1}, \left[ (t_2).Q \right]_{t_2}^{l_1}, \left[ (y).R \right]_{t_3}^{l_1,l_2} \right) \xrightarrow{\nu x.(x).\left[ (t) \right]_{l_2}^{l_1}}
\]
2.5. The proposed sr-calculus

\[
\left( \{t_1/x\}^{t_1,l_1}_{t_1,l_1} \right) \left( \{t_2,R\}^{t_2,R}_{t_2,R} \right) \left( \{t_3/R\}^{t_3/R}_{t_3/R} \right) \rightarrow \left( \{t_1,x\}^{t_1,l_1}_{t_1,l_1} \right) \rightarrow \left( \{t_2/R\}^{t_2,R}_{t_2,R} \right) \rightarrow \left( \{t_3/R\}^{t_3/R}_{t_3/R} \right)
\]

\[
\left( \{t_1/x\}^{t_1,l_1}_{t_1,l_1} \right) \left( \{t_2,R\}^{t_2,R}_{t_2,R} \right) \left( \{t_3/R\}^{t_3/R}_{t_3/R} \right) \rightarrow \left( \{t_1,x\}^{t_1,l_1}_{t_1,l_1} \right) \rightarrow \left( \{t_2/R\}^{t_2,R}_{t_2,R} \right) \rightarrow \left( \{t_3/R\}^{t_3/R}_{t_3/R} \right)
\]

\[
\left( \{t_1/x\}^{t_1,l_1}_{t_1,l_1} \right) \left( \{t_2,R\}^{t_2,R}_{t_2,R} \right) \left( \{t_3/R\}^{t_3/R}_{t_3/R} \right) \rightarrow \left( \{t_1,x\}^{t_1,l_1}_{t_1,l_1} \right) \rightarrow \left( \{t_2/R\}^{t_2,R}_{t_2,R} \right) \rightarrow \left( \{t_3/R\}^{t_3/R}_{t_3/R} \right)
\]

Each labeled transition step is similar as in the previous example with the only difference that each rule is applied twice due to the two broadcast communications.

**Example on modeling the attacker knowledge base**

Let us consider the example topology in the Figure 2.9. Next, I demonstrate how to model the fact that the attacker node collects information, namely, how the attacker knowledge base is extended during the protocol.

![Figure 2.9: In this example scenario, node 1 initiates the route discovery towards node 3, and node A is the attacker which intercepts the requests req1 and req2.](image)

The topology in Figure 2.9 is specified as the extended network \( E_{\text{top}} \):

\[
E_{\text{top}} \overset{\text{def}}{=} \left[ P_1 \right]^{t_1,l_1}_{t_1,l_1} \left[ P_2 \right]^{t_2,l_2}_{t_2,l_2} \left[ P_A \right]^{t_3,l_3}_{t_3,l_3} \left[ P_3 \right]^{t_4,l_4}_{t_4,l_4},
\]

where \( P_1 \), \( P_2 \), \( P_A \), \( P_3 \) correspond to the nodes 1, 2, 3, and A, respectively. After node 1 broadcasts req1, \( E_{\text{top}} \) gets into the state \( E_{\text{top}}' \), where

\[
E_{\text{top}}' \overset{\text{def}}{=} \left\{ \text{req1} / x \right\}^{t_2,l_2}_{t_2,l_2} \left[ P_1 \right]^{t_1,l_1}_{t_1,l_1} \left[ P_2 \right]^{t_2,l_2}_{t_2,l_2} \left[ P_A \right]^{t_3,l_3}_{t_3,l_3} \left[ P_3 \right]^{t_4,l_4}_{t_4,l_4} \left[ P_4 \right]^{t_4,l_4}_{t_4,l_4},
\]

The active substitution with range \( \left\{ \text{req1} / x \right\}^{t_2,l_2}_{t_2,l_2} \) means that the attacker node has intercepted the request req1 due to \( l_2 \in \{ t_2, l_2 \} \). Thus, at this time the knowledge base of the attacker node is increased with req1. \( P_1' \) is the process that we reach from \( P_1 \) after \( P_1 \) broadcasts req1.

Then, the network \( E_{\text{top}}'' \) reaches the state \( E_{\text{top}}'' \) after node 2 has broadcast req2:

\[
E_{\text{top}}'' \overset{\text{def}}{=} \left\{ \text{req2} / y \right\}^{t_1,l_1}_{t_1,l_1} \left\{ \text{req1} / x \right\}^{t_2,l_2}_{t_2,l_2} \left[ P_1 \right]^{t_1,l_1}_{t_1,l_1} \left[ P_2 \right]^{t_2,l_2}_{t_2,l_2} \left[ P_A \right]^{t_3,l_3}_{t_3,l_3} \left[ P_3 \right]^{t_4,l_4}_{t_4,l_4} \left[ P_4 \right]^{t_4,l_4}_{t_4,l_4},
\]

The active substitution with range \( \left\{ \text{req2} / y \right\}^{t_1,l_1}_{t_1,l_1} \) says that the attacker node has intercepted the request req2 because \( l_2 \in \{ t_1, l_1 \} \). Hence, at this point, the knowledge of the attacker node has been extended with req2. Formally, let us assume that the initial knowledge of the attacker is empty, then we have \( \phi(l_a) = \left\{ \text{req2} / y \right\} \cup \left\{ \text{req1} / x \right\} \).
2.6 Analyzing the SRP protocol with the sr-calculus

In this subsection, I demonstrate the usabiliy of the sr-calculus by modeling (the first version of) SRP and the attack shown in Section 2.2.1. The proof technique presented here can be seen as a traditional analyzing technique, which is based on forward search. Namely, we simulate the behavior of the protocol from the beginning until an attack is detected. In most cases, when using a process algebra language to analyze systems and protocols, the authors use this technique [27], [50], [64]. Unfortunately, this proof technique cannot work in case of more complicated systems and protocols. Therefore, in Section 2.7, I propose a proof technique that combines the backward deduction approach with the Definition 7, which is more systematic and efficient for source routing protocols.

The scenario in Section 2.2.1 is modeled by the extended network defined as:

\[ E_{sr}^{real} \overset{\text{def}}{=} [\{P_1\}_{l_1}^{t_1, l_n}] | [\{P_2\}_{l_2}^{t_2}] | [\{P_A\}_{l_3}^{t_1, l_2, t_3, l_4}] | [\{P_B\}_{l_4}^{t_4}] | [\{P_C\}_{l_5}^{t_5}], \]

where the description of the nodes in the parallel compositions corresponds to \( N_1, N_2, A, N_3, \) and \( N_4 \), respectively, in Figure 2.2.

I use the following functions: The function \( \text{mac}(t_1, t_2) \) computes the message authentication code of the message \( t_1 \) using the secret key \( t_2 \). The shared key between the nodes \( l_i \) and \( l_j \) is modeled by the function \( k(l_i, l_j) \). The function \( \text{list}(l_1, \ldots, l_n) \) models the list of node IDs, and \( \text{list}() \) models the empty ID list. Functions \( \text{prefix}(\text{List}, l_i) \) and \( \text{next}(\text{List}, l_i) \) return the element right before and after \( l_i \) in the list \( \text{List} \), respectively; they return \( \text{undef} \) if there is no any element before or after \( l_i \). Function \( \text{toend}(\text{List}, l_i) \) that appends \( l_i \) to the end of \( \text{List} \). Functions \( \text{fst}(\text{List}) \) and \( \text{lst}(\text{List}) \) represent the first, and last element of \( \text{List} \), respectively. Function \( i((l_1, \ldots, l_n)) \) returns the \( i \)-th \((i \in \{1, \ldots, n\})\) element \( l_i \) of the tuple \((l_1, \ldots, l_n)\).

Note that in case of ad-hoc network, generally the behaviour of nodes can be specified in the same way that every node can be a source, an intermediate node, and a destination node. However, for the sake of brevity, below I specifically consider the scenario in Section 2.2.1 and define the behaviour of nodes according to the scenario. This is sufficient to prove the vulnerability of (the first version of) SRP, under the assumption that there is no neighbor-to-neighbor authentication. The operation of the source node is specified as follows (for simplicity, I omit the presence of sequence number and message ID in messages, but note that this attack works in the same way in the presence of them):

\[
\begin{align*}
P_1 & \overset{\text{def}}{=} \text{let } \text{MAC}_{13} = \text{mac}(l_1, l_3, k(l_1, l_3)) \text{ in Init.} \\
\text{Init} & \overset{\text{def}}{=} \langle (\text{req}, l_1, l_3, \text{MAC}_{13}, \text{list}()) \rangle \text{!Rep}. \\
\text{Rep}_1 & \overset{\text{def}}{=} (x_{req}, 1) [2(x_{req}) = l_1] [3(x_{req}) = l_3] [4(x_{req}) = l_3] [\text{fst}(5(x_{req})) \in \{l_2, l_n\}] \\
& \quad [\text{mac}(l_1, l_3, 5(x_{req})) \land k(l_1, l_3)] = 6(x_{req}) [\langle \text{Accept} \rangle . \text{nil}] \\
\end{align*}
\]

Intuitively, the first row models that the node \( l_1 \) computes the MAC using the key it shares with the destination node \( l_3 \). The second row means that node \( l_1 \) generates the route request message that includes the ID of the source and the target nodes, and the message authentication code \( \text{MAC}_{13} \), then \( l_1 \) broadcasts it and waits for the reply. The exclamation mark models the infinite replication of \( \text{Rep}_1 \). Finally, the third row means that when \( l_1 \) receives a message, it checks whether (i) it is the addressse, (ii) the message is a reply, (iii) the ID of the source and the target nodes, and (iv) the message authentication code are correct. If all are correct then it signals the special constant \( \text{Accept} \). The description of the process \( P_2 \) is the following:

\[
\begin{align*}
P_2 & \overset{\text{def}}{=} (y_{req}, 1) [1(y_{req}) = \text{req}] [2(y_{req}) = \text{req}] [3(y_{req}) = \text{req}] [4(y_{req}) = \text{req}] [\text{toend}(5(y_{req}), l_2)) \text{!Rep}. \\
\text{Rep}_2 & \overset{\text{def}}{=} (y_{req}, 1) [1(y_{req}) = l_2] [2(y_{req}) = y_{req}] [3(y_{req}) = \text{req}] [4(y_{req}) = \text{req}] [5(y_{req}) = \text{req}] [6(y_{req}) = \text{req}] [\langle \text{Accept} \rangle . \text{nil}] \\
\end{align*}
\]
2.6. Analyzing the SRP protocol with the sr-calculus

Intuitively, on receiving a message node $l_2$ checks if it is a request, if so then node $l_2$ appends its ID to the end of the ID list, then re-broadcasts the request and waits for a reply. When it receives the reply message it checks (i) if the message is intended to it, (ii) the message is a reply, (iii) the ID next to $l_2$ in the list corresponds to the neighbors of node $l_2$ and forwards the reply to the source node $l_1$ if all verification steps pass.

Finally, the operation of the destination node is modeled as:

\[
P_3 \overset{\text{def}}{=} (z_{\text{req}}, \text{nil} \rightarrow z_{\text{req}}) \cdot [1(z_{\text{req}}) = \text{req}] \cdot [3(z_{\text{req}}) = l_3] \cdot [\text{mac}((2(z_{\text{req}}), 3(z_{\text{req}})), k(l_1, l_3)) = 4(z_{\text{req}})].
\]

let $\text{MAC}_{31} = \text{mac}((1(z_{\text{req}}), 2(z_{\text{req}}), 3(z_{\text{req}}), 5(z_{\text{req}})), k(l_1, l_3))$ in

let $l_{\text{prev}} = \text{lst}(5(z_{\text{req}}))$ in $(l_{\text{prev}}, \text{rep}, 2(z_{\text{req}}), 3(z_{\text{req}}), 5(z_{\text{req}}), \text{MAC}_{31}).\text{nil}$

Intuitively, on receiving a message node $l_3$ checks (i) if the message is a request, and (ii) it is the destination, and verifies the MAC embedded in the request using its shared key with $l_1$. If so then node $l_3$ creates a reply message and sends it back to the last node in the list.

The description of the node $l_4$ is the same as the node $l_2$ with the difference that it appends $l_4$ to the list instead of $l_2$. I refer the reader to [TH05, 2010] for details.

I give the model $(M_A)$ of the attacker node as follows: Assume that the attacker cannot forge the message authentication codes $\text{MAC}_{13}$ and $\text{MAC}_{31}$ without possessing correct keys. Initially, the attacker node knows the IDs of its neighbors $\{l_1, l_2, l_3, l_4\}$. The attacker can create new data $n$, and can append elements of $\{l_1, l_2, l_3, l_4\}$, and $n$ to the end of the ID list it receives. Finally, it can broadcast and unicast its messages to honest nodes.

The attacker overhears only messages sent by its neighbors. The attacker combines this accumulated knowledge and its initial knowledge to perform attacks. Let $L_{\text{in}}$ be a tuple that consists of the elements in $\{l_1, l_2, l_3, l_4\}$.

Formally, the operation of the attacker node is defined as follows: $P_A \overset{\text{def}}{=} (\check{x}) \cdot \nu n. (f(\check{x}, T_{\text{in}}, n))$, where $\check{x}$ is a tuple $(x_1, \ldots, x_n)$ of variables, $\nu n$ means that the attacker creates new data $n$. The function $f(\check{x}, T_{\text{in}}, n)$ represents the message the attacker generates from (i) the eavesdropped messages that it receives by binding them to $\check{x}$, (ii) its initial knowledge and (iii) the newly generated data $n$, respectively.

In order to apply the Definition 7 in my proof, I define the ideal version of $E_{\text{SRP}}^\text{real}$, denoted as $E_{\text{SRP}}^\text{ideal}$. The definition of $E_{\text{SRP}}^\text{ideal}$ is the same as $E_{\text{SRP}}^\text{real}$ except that the desription of $N_1$ is $\{P_{\text{ideal}}(\{l_2, l_3\})\}$. Process $P_{\text{ideal}}$ models the ideal operation of the source node $N_1$ in the sense that although the source node does not know the route to the destination it is equipped with the special function $\text{consistent(List)}$ that informs it about the correctness of the returned route.

I define the ideal behavior of the source node as follow:

\[
P_{\text{ideal}} \overset{\text{def}}{=} \text{let } \text{MAC}_{13} = \text{mac}((l_1, l_3), k(l_1, l_3)) \text{ in } \text{Init}_{\text{ideal}}.
\]

\[
\text{Init}_{\text{ideal}} \overset{\text{def}}{=} (\text{req}, l_1, l_3, \text{MAC}_{13}, \text{lst}()) \cdot \text{Rep}_{\text{ideal}}.
\]

\[
\text{Rep}_{\text{ideal}} \overset{\text{def}}{=} (x_{\text{req}}, \text{nil} \rightarrow x_{\text{req}}) \cdot [1(x_{\text{req}}) = \text{req}] \cdot [3(x_{\text{req}}) = l_3] \cdot [\text{mac}((l_1, l_3, 5(x_{\text{req}})), k(l_1, l_3)) = 6(x_{\text{req}})] \cdot [\text{consistent}(5(x_{\text{req}})) = \text{true}] \cdot (\text{Accept}) \cdot \text{nil}
\]

Intuitively, in the ideal model, every route reply that contains a non-existent route is caught and filtered out by the initiator of the route discovery. Therefore, security is achieved by definition.

Theorem 1. The SRP protocol is insecure.

Proof. (Sketch) Note again that the attack below does not take into consideration the neighbor-to-neighbor authentication during route discovery [55]. I will show that $E_{\text{SRP}}^\text{real} \approx_{\text{N}} E_{\text{SRP}}^\text{ideal}$ does not hold in the presence of the attacker $M_A$ because the third condition of the Definition 5 is violated. When the attacker node receives the request message $(r_{\text{req}}, l_1, l_3, \text{MAC}_{13}, [l_2])$ from node $l_2$, it creates some new fake node indentifier $n$, then it adds $n$ and the identifier $l_4$ to the list $[l_2]$, then
it re-broadcasts ($rreq$, $l_1$, $l_3$, $MAC_{13}$, $l_2$, $n$, $l_4$). When this message reaches the target node $l_3$ it passes all the verifications made by $l_3$. Then, node $l_3$ generates the reply ($l_4$, $rrep$, $l_1$, $l_3$, $MAC_{31}$) and sends it back to $l_4$. The attacker node overhears this message and forwards it to the source $l_1$ in the name of $l_2$. As the result, in $E_{srp}^{real}$ node $l_1$ accepts the returned invalid route $[l_2$, $n$, $l_4]$ and outputs Accept by the $\nu x.\tau_l^{(24)}$ transition step. However, in $E_{srp}^{ideal}$ node $l_1$ does not accept the returned route, thus, Accept is not output. Formally, at this point $E_{srp}^{ideal}$ cannot perform the transition $\nu x.\tau_l^{(24)}$, which violates the third condition of Definition 5.

2.7 BDSR: The backward deduction algorithm for source routing protocols

I give the name BDSR for the algorithm, which contains the first letters of the words Backward, Deduction, and Source Routing. I develop a systematic proof technique, that enables us to reason about the security of routing protocols in an efficient way. This proof technique is based on backward deduction, namely, I start with the assumption that the source has accepted an invalid route, and based on the definition of the protocol we reason backward step-by-step to find out how this could have happened. In case we get a contradiction it means that the starting assumption must not be valid, and the protocol is secure. The main advantage of the BDSR is that it does not require the definition of specific network topology, hence, we need only to specify the behavior of the source, the destination and the intermediate node, providing that intermediate nodes have an uniform behavior.

Compared to the forward search approach, the method based on backward reasoning has the advantage that the verification can be made on an arbitrary topology and a strong attacker computation ability can be assumed. Using backward reasoning in [62] the authors have been succeeded in verifying the occurrence of loop in mobile ad-hoc networks assuming an arbitrary topology. The authors used graph transformation, which is suitable for modeling pure routing protocols without cryptography, however, it is not well-suited for modeling secure routing protocols. Unlike the this method, my proposed method is specifically designed for analyzing secure routing protocols, which cannot be solved with those related works.

The basic idea of my approach is that initially an invalid route $r$, which is represented by an ID list $[List_{invalid}]$ of different IDs, is supposed to be accepted at the end of the route discovery. The task of the verification algorithm is to confirm this assumption by finding a sequence of message exchanges along with a topology in which both the destination and source accept $[List_{invalid}]$ as a valid route, or to give a refutation in case no attack scenario can be found. To do this, we follow the way of the reply that contains $[List_{invalid}]$, in a backward manner. The possible paths of this reply are investigated by reasoning about the nodes and edges through which this reply and the corresponding request must have traversed during the route discovery. On searching for the possible paths of the reply and request backward, whenever an attacker node is reached, it means that the reply or request has been forwarded (and may be modified) by the attacker node. At this point, based on the reply/request we are aware of the information about which message should the attacker forward so that it will be accepted later, that is, which messages should the attacker generate in order to perform a successful attack. This is then followed by examining how the attacker could generate these messages based on its knowledge.

The attackers are able to compose a reply or a request message using its computational ability and knowledge base. Note that while the computation ability of the attackers is fixed, their knowledge base is continually updated during the route discovery session. Hence, by backward reasoning we mean the reasoning about three issues: (i) Can the attacker generate each part of the message based only on its computational ability and knowledge base? (ii) Which messages should the attack node intercept in case it cannot set up a whole reply/request based solely on its computational ability and knowledge? (iii) How the topology should have been formed such that the attacker is able to intercept or receive the required message parts?
During the backward deduction, we keep track of the topology $T_{top}$, in which a given attack scenario (if any) is feasible. At the beginning, $T_{top}$ is empty, and it is iteratively updated with new edges and nodes during the backward reasoning. I mainly consider bi-directional edges. I define the set of the edges in the route $[List_{invalid}]$, denoted by $T_{invalid}$.

The initial state, $state_{accept}$, of the backward deduction is the state in which $[List_{invalid}]$ is accepted at the end of a route discovery session. The terminal state, $state_{reqinit}$, is the state in which the source node has initiated the route discovery by broadcasting the request. The backward reasoning ends when we have reached back to the terminal state. There can be two possibilities: (1) We proved our initial assumption that $[List_{invalid}]$, which is accepted at the end, is indeed an invalid route; (2) we refuted the initial assumption, and proved that $[List_{invalid}]$ cannot be an invalid route. In the first case, an attack scenario is detected, and can happen if during the backward deduction procedure, $T_{invalid}$ will never be a subset of $T_{top}$. In the second case, the deduction can only terminate when $T_{invalid} \subseteq T_{top}$ holds. Intuitively, whenever the source accepts an ID list at the end of a route discovery session, the list must represent a valid route.

The backward deduction procedure can be illustrated as a derivation tree (on the left side of Figure 2.10), in which the state $state_{accept}$ is the root. Nodes represent the states (a set of message exchanges and current topology) we reach during the backward reasoning. The edges represent deduction steps leading us from one state to another. The leaves of the tree are the terminal states, $state_{reqinit}$. The backward deduction is composed of several deduction branches. The main goal of the deduction is to find an attack, hence, a deduction branch gets stuck whenever we get $T_{invalid} \subseteq T_{top}$ after updating $T_{top}$. In this case, we return to the last branching point, and follow an another deduction branch. In the “honest” phases $Ph-H1$, $Ph-H2$ we reason about the message exchanges between honest nodes. In $Ph-H1$, we investigate how the reply that contains $[List_{invalid}]$ should propagate during the route discovery. Whenever we deduce that at a given state the message should be sent by an attacker node, we step into the “attacker” phase $Ph-A$. In $Ph-A$, we reason about how the attacker could generate a request or a reply, which leads to a successful attack. There can be two cases: the attacker is able to compute the whole reply/request message, based only on its computation ability and the current knowledge base. In this case, an attack scenario is detected. Otherwise, if some part $v_i$ of the reply/request cannot be generated directly by the attacker, the verification is continued with phase $Ph-H2$, where we examine how the attacker can receive or intercept a message that contains $v_i$. According to the observation discussed in Section 2.4, the attacker must pass on replies in order to perform a successful attack. Hence, the first phase $Ph-A$ contains at least one state, while the rest attacker phases may be skipped. The states in the honest and attacker phases are denoted by the indices $hon_i$ and $att_i$, respectively, for different values of $i$.

The phase $Ph-Rec$ represents a “recursive” application of the attacker phase $Ph-A$. Typically, if $(N = 1)$ then there is one attacker node and we consider its interference in the reply phase. The case $(N \geq 2)$ takes into account the possibility of several attackers and interleaving sessions, or one attacker node who interferes in both the request and reply directions.

**The attacker phase $Ph-A$:** In the attacker phase $Ph-A$, if we found that the attacker must have sent the reply $t_{attrep}$ or the request $t_{attreq}$ to be successful, then in the rest of the steps we deduce how this message could have been composed. We let both the reply and request have the form $(head; v_1; \ldots; [List]; \ldots; v_k)$, which is true in most source routing protocols. The head part, $head$, is the tuple $(rreq/rrep : T_{req/rrep}, s : T_{str}, d : T_{str}, sID : T_{str})$, where the first element has REQ/REP type, the remaining three elements have string type. The reason for giving the string type instead of node ID type or name type for $s$, $d$, and $sID$, is that we want to allow the attackers to replace these elements with some data of other types. The list $[List]$ is also given a string type, while $v_1, \ldots, v_n$ are additional protocol specific parts, which typically are some (crypto) functions computed on one portion of the message. For instance, in case of the SRP protocol we have one function part, which is the MAC, while the Ariadne protocol includes hash and digital signature functions.

During the backward reasoning we attempt to find attacks with minimal number of transition steps. The general steps of the BDSR algorithm in $Ph-A$ is as follows:
2. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WIRELESS AD-HOC ROUTING PROTOCOLS

Figure 2.10: The figure illustrates the main phases of the backward deduction procedure. The circles and asterisks represent the particular states during the deduction procedure. The states with the index accept and reqinit represent the initial and terminal states, respectively. The general derivation tree can be found on the left and the corresponding tree for the proof technique of the sr-calculus is on the right. The dots after an arrow between each node pair represent a set of edges, which can be empty, meaning that the two nodes are the same.

1. First of all, we examine whether the attack could be performed when the attacker has forwarded the request/reply \( t_{\text{req}/\text{rep}} \) by following to the protocol correctly. If following this deduction branch, the deduction terminates such that \( T_{\text{invalid}} \not\subseteq T_{\text{top}} \), then an attack scenario is detected. The intuition behind this step is that there can be an attack where the attackers do not modify/forge both the reply and request, but only one of them.

2. Otherwise, if the deduction in the first point can terminate only with \( T_{\text{invalid}} \subseteq T_{\text{top}} \), we return to examine how the attacker can obtain or compute each part of \( t_{\text{attreq}/\text{attrep}} \), where \( t_{\text{attreq}/\text{attrep}} = (\text{head}; v_1; \ldots; [\text{List}]; \ldots; v_k) \). An attack scenario is detected only when every element of \( t_{\text{attreq}/\text{attrep}} \) can be computed/obtained by the attacker(s).

I define priority/weight on data of different types. I distinguish three classes of priority. The keyed crypto functions such as digital signatures, MAC function, public and symmetric key encryption have the highest priority. The next highest priority class is assigned to keyless crypto functions such as one-way hash function. The lowest priority is given to non-crypto functions and data construct such as \( rrep \) and \( rreq \), node IDs, . Within the same priority, data are classified by weight, which specifies the number of variables and constants in them. The more sub-elements (names, constants, variables, functions) are included in a data construct the larger its weight is.

Let \( W \) be a set that contains the message parts of a request/reply to be examined, whether they can be computed or obtained by the attackers. To analyze how the attacker can compose the reply/request, \( t_{\text{attreq}/\text{attrep}} \), we perform the following steps. Note that here I only give an overview, the more detailed steps within each point below will be provided later.

(I.) First, \( t_{\text{attreq}/\text{attrep}} \) is decomposed and the resulted parts \( \text{head}, v_1, \ldots, [\text{List}], v_k \) are put into \( W \).

(II.) If there still are unexamined terms in \( W \) we choose one of the highest priority group of terms, and within this group we start with the term that has highest weight that has not been examined before.

(III.) Let us denote the attacker’s knowledge base by \( K_{\text{att}} \). For the elemental data \( t_n \) such as constant and names, we check whether \( t_n \in K_{\text{att}} \). For each function \( t_f \) we examine if the attacker can compute it using the current knowledge base (e.g., keys for crypto functions). If
In this section, I provide a proof technique for the BDSR algorithm and the definition of labeled bisimilarity. The main advantage of this proof technique is that it combines the precise mathematical background of the BDSR algorithm and the definition of labeled bisimilarity. The main advantage of this proof technique is that it combines the precise mathematical background of the BDSR algorithm and the definition of labeled bisimilarity. The main advantage of this proof technique is that it combines the precise mathematical background of the BDSR algorithm and the definition of labeled bisimilarity. The main advantage of this proof technique is that it combines the precise mathematical background of the BDSR algorithm and the definition of labeled bisimilarity.

2.7.1 Applying BDSR for the sr-calculus

In this section, I provide a proof technique for the sr-calculus, which is based on the combination of the BDSR algorithm and the definition of labeled bisimilarity. The main advantage of this proof technique is that it combines the precise mathematical background of the BDSR algorithm and the definition of labeled bisimilarity.

For applying BDSR, I define an ideal and a real system a bit differently compared to the systems presented by an extended network, namely, \( E_{\text{accept}(t_{\text{list}})} \) is the extended network corresponding to state \( \text{accept} \), while \( E_{\text{reqinit}} \) corresponds to state \( \text{reqinit} \). Each edge from a state \( E_i \) to another state \( E_j \) represents a corresponding labeled transition \( E_i \xrightarrow{\alpha} E_j \), where \( \alpha \) can be a broadcast send/receive, a unicast send/receive and a silent action. States \( \text{state}_{\text{hon}} \) and \( \text{state}_{\text{att}} \) are represented by an extended network \( E_{\text{hon}} \) and \( E_{\text{att}} \), which get from \( E_{\text{accept}(t_{\text{list}})} \) after performing a series of labeled transitions in a backward phase.

For applying BDSR, I define an ideal and a real system a bit differently compared to the systems presented by an extended network, namely, \( E_{\text{accept}(t_{\text{list}})} \) is the extended network corresponding to state \( \text{accept} \), while \( E_{\text{reqinit}} \) corresponds to state \( \text{reqinit} \). Each edge from a state \( E_i \) to another state \( E_j \) represents a corresponding labeled transition \( E_i \xrightarrow{\alpha} E_j \), where \( \alpha \) can be a broadcast send/receive, a unicast send/receive and a silent action. States \( \text{state}_{\text{hon}} \) and \( \text{state}_{\text{att}} \) are represented by an extended network \( E_{\text{hon}} \) and \( E_{\text{att}} \), which get from \( E_{\text{accept}(t_{\text{list}})} \) after performing a series of labeled transitions in a backward phase.

For applying BDSR, I define an ideal and a real system a bit differently compared to the systems presented by an extended network, namely, \( E_{\text{accept}(t_{\text{list}})} \) is the extended network corresponding to state \( \text{accept} \), while \( E_{\text{reqinit}} \) corresponds to state \( \text{reqinit} \). Each edge from a state \( E_i \) to another state \( E_j \) represents a corresponding labeled transition \( E_i \xrightarrow{\alpha} E_j \), where \( \alpha \) can be a broadcast send/receive, a unicast send/receive and a silent action. States \( \text{state}_{\text{hon}} \) and \( \text{state}_{\text{att}} \) are represented by an extended network \( E_{\text{hon}} \) and \( E_{\text{att}} \), which get from \( E_{\text{accept}(t_{\text{list}})} \) after performing a series of labeled transitions in a backward phase.

For applying BDSR, I define an ideal and a real system a bit differently compared to the systems presented by an extended network, namely, \( E_{\text{accept}(t_{\text{list}})} \) is the extended network corresponding to state \( \text{accept} \), while \( E_{\text{reqinit}} \) corresponds to state \( \text{reqinit} \). Each edge from a state \( E_i \) to another state \( E_j \) represents a corresponding labeled transition \( E_i \xrightarrow{\alpha} E_j \), where \( \alpha \) can be a broadcast send/receive, a unicast send/receive and a silent action. States \( \text{state}_{\text{hon}} \) and \( \text{state}_{\text{att}} \) are represented by an extended network \( E_{\text{hon}} \) and \( E_{\text{att}} \), which get from \( E_{\text{accept}(t_{\text{list}})} \) after performing a series of labeled transitions in a backward phase.

For applying BDSR, I define an ideal and a real system a bit differently compared to the systems presented by an extended network, namely, \( E_{\text{accept}(t_{\text{list}})} \) is the extended network corresponding to state \( \text{accept} \), while \( E_{\text{reqinit}} \) corresponds to state \( \text{reqinit} \). Each edge from a state \( E_i \) to another state \( E_j \) represents a corresponding labeled transition \( E_i \xrightarrow{\alpha} E_j \), where \( \alpha \) can be a broadcast send/receive, a unicast send/receive and a silent action. States \( \text{state}_{\text{hon}} \) and \( \text{state}_{\text{att}} \) are represented by an extended network \( E_{\text{hon}} \) and \( E_{\text{att}} \), which get from \( E_{\text{accept}(t_{\text{list}})} \) after performing a series of labeled transitions in a backward phase.

For applying BDSR, I define an ideal and a real system a bit differently compared to the systems presented by an extended network, namely, \( E_{\text{accept}(t_{\text{list}})} \) is the extended network corresponding to state \( \text{accept} \), while \( E_{\text{reqinit}} \) corresponds to state \( \text{reqinit} \). Each edge from a state \( E_i \) to another state \( E_j \) represents a corresponding labeled transition \( E_i \xrightarrow{\alpha} E_j \), where \( \alpha \) can be a broadcast send/receive, a unicast send/receive and a silent action. States \( \text{state}_{\text{hon}} \) and \( \text{state}_{\text{att}} \) are represented by an extended network \( E_{\text{hon}} \) and \( E_{\text{att}} \), which get from \( E_{\text{accept}(t_{\text{list}})} \) after performing a series of labeled transitions in a backward phase.

For applying BDSR, I define an ideal and a real system a bit differently compared to the systems presented by an extended network, namely, \( E_{\text{accept}(t_{\text{list}})} \) is the extended network corresponding to state \( \text{accept} \), while \( E_{\text{reqinit}} \) corresponds to state \( \text{reqinit} \). Each edge from a state \( E_i \) to another state \( E_j \) represents a corresponding labeled transition \( E_i \xrightarrow{\alpha} E_j \), where \( \alpha \) can be a broadcast send/receive, a unicast send/receive and a silent action. States \( \text{state}_{\text{hon}} \) and \( \text{state}_{\text{att}} \) are represented by an extended network \( E_{\text{hon}} \) and \( E_{\text{att}} \), which get from \( E_{\text{accept}(t_{\text{list}})} \) after performing a series of labeled transitions in a backward phase.

For applying BDSR, I define an ideal and a real system a bit differently compared to the systems presented by an extended network, namely, \( E_{\text{accept}(t_{\text{list}})} \) is the extended network corresponding to state \( \text{accept} \), while \( E_{\text{reqinit}} \) corresponds to state \( \text{reqinit} \). Each edge from a state \( E_i \) to another state \( E_j \) represents a corresponding labeled transition \( E_i \xrightarrow{\alpha} E_j \), where \( \alpha \) can be a broadcast send/receive, a unicast send/receive and a silent action. States \( \text{state}_{\text{hon}} \) and \( \text{state}_{\text{att}} \) are represented by an extended network \( E_{\text{hon}} \) and \( E_{\text{att}} \), which get from \( E_{\text{accept}(t_{\text{list}})} \) after performing a series of labeled transitions in a backward phase.
As mentioned before, the backward deduction is based on performing labeled transitions backward from $E_{\text{accept}(t_{\text{list}})}$ to $E_{\text{reqinit}}$. I use the upper index real ($E_{\text{reqinit}}^{\text{real}}$, $E_{\text{accept}(t_{\text{list}})}^{\text{real}}$) and ideal ($E_{\text{reqinit}}^{\text{ideal}}$, $E_{\text{accept}(t_{\text{list}})}^{\text{ideal}}$) to denote the corresponding network states in the real and ideal systems, respectively. The backward deduction is based on backward application of labeled transitions from $E_{\text{accept}(t_{\text{list}})}$ to $E_{\text{reqinit}}$:

$$E_{\text{accept}(t_{\text{list}})}^* \leftarrow a_1^* \leftarrow \ldots \leftarrow a_n^* \leftarrow E_{\text{reqinit}},$$

where $a_1, \ldots, a_n$ can be a broadcast, an unicast, or a receive action. For instance, the following backward deduction trace shows a possible scenario about how we can get from the state when the destination sends a reply, to the state when the source has sent the corresponding request:

$$E_{\text{dstsentREP}}^{\nu_2(z)}: \frac{\nu_2(z) \leftarrow \text{last-TRANS}}{t_{\text{int}} \ldots} \leftarrow E_{\text{dstrecvdREQ}}^{t_{\text{req}}: \nu_1^*: \nu_2^*: \nu_3^* \in \nu_2^*: \nu_2^*: \nu_3^*} \leftarrow E_{\text{intsentREQ}}$$

where the frame of $E_{\text{dstsentREP}}$ contains the substitution $\{t_{\text{req}}/z\}^{\nu_1^*}$, and the frames of $E_{\text{intsentREQINIT}}$ contains $\{t_{\text{req}}/y\}^{\nu_2^*}$. $\nu_1^*$ and $\nu_2^*$ are the neighborhood of $t_{\text{int}}$ and $t_{\text{src}}$, respectively. Intuitively, the trace

$$E_{\text{dstsentREP}}^{\nu_2(z)}: \frac{\nu_2(z) \leftarrow \text{last-TRANS}}{t_{\text{int}} \ldots} \leftarrow E_{\text{dstrecvdREQ}}^{t_{\text{req}}: \nu_1^*: \nu_2^*: \nu_3^* \in \nu_2^*: \nu_2^*: \nu_3^*} \leftarrow E_{\text{intsentREQ}}$$

says that in order to $t_{\text{int}}$ can send the reply $t_{\text{req}}$, before this, it should receive the request $t_{\text{req}}$ from $t_{\text{int}}$. The trace

$$E_{\text{intsentREQ}}^{\nu_2(y)}: \frac{\nu_2(y) \leftarrow \text{last-TRANS}}{t_{\text{int}} \ldots} \leftarrow E_{\text{intrecvdREQ}}^{(t_{\text{req}})^{\nu_1^*: \nu_2^*: \nu_3^*} \in \nu_1^*: \nu_2^*: \nu_3^*} \leftarrow E_{\text{reqinit}}$$

says that in order to node $t_{\text{int}}$ can send the request $t_{\text{req}}$, it should receive the request $t_{\text{req}}$ from $t_{\text{int}}$. The whole example trace corresponds to the following scenario: On the route $t_{\text{src}} - t_{\text{int}} - t_{\text{dst}}$, (1) $t_{\text{src}}$ broadcasts $t_{\text{reqinit}}$; (2) $t_{\text{int}}$ received $t_{\text{reqinit}}$, performs calculations, and broadcasts $t_{\text{req}}$; (3) $t_{\text{dst}}$ received $t_{\text{req}}$, performs verifications, and returns $t_{\text{req}}$.

In order to perform a systematic proof based on Definition 5, I distinguish the ideal system and the real system in the following way: In the ideal system, the source always can check the correctness of the returned route $t_{\text{list}}$ by using the special function $\text{consistent}(t_{\text{list}})$, and only outputs $\text{accept}(t_{\text{list}})$ if $t_{\text{list}}$ is a correct route from the source to the destination. To attain this, I define the ideal system such that the backward deduction can only terminate without finding an attack. To apply this in the BDSR algorithm, in the ideal system, for such deductions where we can get back to the $E_{\text{reqinit}}$ such that $T_{\text{invalid}} \notin T_{\text{top}}$, the last transition

$$E_{\text{intrecvdREQ}}^{(t_{\text{reqinit}})^{\nu_2^*: \nu_3^*} \in \nu_1^*: \nu_2^*: \nu_3^*} \leftarrow E_{\text{reqinit}}$$

is forbidden, which models the receiving of the initial request, to be performed in the last deduction branch. Meanwhile, this is allowed in the real system, because I give the possibility for invalid route to be accepted. Hence, in case an attack is detected in the real system, the ideal and the real system can be distinguished based on this last transition.

To prove the security of on-demand source routing protocols based on the backward deduction approach, I apply the Definition 5 in a reverse phase, specifically:

**Definition 8.** Let $E_{\text{accept}(t_{\text{list}})}^{\text{real}}$ and $E_{\text{accept}(t_{\text{list}})}^{\text{ideal}}$ be the real and the ideal specification variants of a (on-demand source) routing protocol $\text{Prot}$ in the sr-calculus. The protocol Prot is said to be secure if for all the possible routes represented by the list $t_{\text{list}}$, the following holds:

1. $E_{\text{accept}(t_{\text{list}})}^{\text{real}} \approx E_{\text{accept}(t_{\text{list}})}^{\text{ideal}}$;
2.7. BDSR: The backward deduction algorithm for source routing protocols

2.7.2 General specification of on-demand source routing protocols

In this subsection, a general and simplified specification of the on-demand source routing protocols is given, which is well-suited for the backward deduction technique. The specification is based on the sr-calculus, but instead of defining specific network topologies, I provide a general specification that includes the specification of a source, a destination, and some intermediate nodes, regardless of the topology.

On-demand source routing protocols have an important flavour that each node usually has the same uniform internal operation: during route discovery each node can play a role of a source node, or an intermediate node, or a destination node. Leveraging this beneficial characteristic, we need to specify only the operation of three nodes instead of all of the nodes in the network, which is more comfortable.

\[ E_{\text{routing}} \overset{\text{def}}{=} \{ \langle P_{\text{src}} \rangle^\sigma_{\text{src}} | i \in 1, \ldots, n \} | \{ P_{\text{int}}^i \}^\sigma_{\text{int}} | \{ P_{\text{dst}} \}^\sigma_{\text{dst}}, \]

where \( N_{\text{src}} = \{ \langle P_{\text{src}} \rangle^\sigma_{\text{src}}, N_{\text{int}}, N_{\text{int}}^i = \{ P_{\text{int}}^i \}^\sigma_{\text{int}}, \) represents the i-th intermediate node, and \( \prod_{i = 1}^n \{ P_{\text{int}}^i \}^\sigma_{\text{int}} \) represents the parallel composition of n intermediate nodes \( \{ P_{\text{int}}^i \}^\sigma_{\text{int}}, \) for \( i \in \{1, \ldots, n\}, N_{\text{dst}} = \{ P_{\text{dst}} \}^\sigma_{\text{dst}} \).

Processes \( P_{\text{src}}, P_{\text{int}}, P_{\text{dst}} \) model the operation of honest nodes. We do not need to include explicitly the behavior of the attacker node(s). The attackers are modeled by the wireless environment in an implicit way, which can be seen as a cooperation of several attackers. In case an attacker scenario is detected, the specific place of the attacker(s) is determined based on the messages it (they) intercepts or sends during the scenario. The number of the intermediate nodes, \( n, \) is also determined based on the specific detected attack scenario. Note that the specific structure of each process depends on the specific routing protocol.

At first, \( T_{\text{top}} \) does not contain any edges, and first state \( E_{\text{accept}(t_{\text{int}})} \) is

\[ E_{\text{accept}(t_{\text{int}})} \overset{\text{def}}{=} \{ \text{accept}(t_{\text{int}}) | x_{\text{accept}} \}^\sigma_{\text{src}} | \{ x_{\text{initknown} 1} / x_{\text{initknown} 1}, \ldots, x_{\text{initknown} 1} \}^\sigma_{\text{int}} \} | \{ P_{\text{src}}^\sigma_{\text{src}} | i \in 1, \ldots, n \} | \{ P_{\text{int}}^i \}^\sigma_{\text{int}} | \{ P_{\text{dst}} \}^\sigma_{\text{dst}}. \]
2. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WIRELESS AD-HOC ROUTING PROTOCOLS

where \( \sigma_{src} \), \( \sigma_{int} \), and \( \sigma_{dst} \) are empty. During the backward deduction, if at one point we found that to make the source accepts \( l_{list} \), the source \( l_{src} \) should send a message \( t_{req/rep} \) to \( l_t \), then we update \( T_{top} \) with the new edge between \( l_{src} \) and \( l_t \). Further, we add \( l_t \) to the neighborhood of \( l_{src} \), \( \sigma_{src} \), and add \( l_{src} \) to \( \sigma_{t} \) (assuming bi-directional edges). \{accept\( (t_{list})/x_{accept}\}\}^{\sigma_{src}} says that the output \( \text{accept}(t_{list}) \) is available to the neighborhood of \( l_{src} \) and the attacker. We add \( l_{a} \) to each substitution, because by default we assume that the attacker nodes can be everywhere in the network. The exact number and location of the attackers depends on the deduction path. Similarly, the terminal state \( E_{reqinit} \) is specified as follows:

\[
E_{reqinit} \equiv \{ \text{accept}(t_{list})/x_{accept}\}^{\sigma_{src}}, \ldots, \{ t_{reqinit}/x_{reqinit}\}^{\sigma_{src}} \mid \{ t_{attinitknowu}/x_{initknowu}, \ldots, \}
\]

\[
t_{attinitknowu}/x_{initknowu}\}_{\sigma_{src}} \mid [P_{src} | P_{reqinit}]^{\sigma_{src}} \mid \prod_{l_{src}} | P_{int} | | \prod_{l_{src}}^{\sigma_{int}} |[P_{dst}]^{\sigma_{dst}}.
\]

In \( E_{reqinit} \), the frame is extended with the substitution \{ \text{t_{reqinit}}/x_{reqinit}\}^{\sigma_{src}} \), and \( P_{reqinit}^{src} \) is the process we get after \( t_{reqinit} \) has been broadcast in \( P_{src} \).

2.7.3 Analyzing the security of Ariadne based on the BDSR algorithm

In this subsection, I analyze the security of the Ariadne protocol using the backward deduction technique. I start with defining the functions one-way hash and digital signature used by Ariadne.

Digital signature schemes rely on pairs of public and secret keys. In each pair, the secret key serves for computing signatures and the public key for verifying those signatures. I introduce two new unary function symbols \( pk \) and \( sk \) for generating public and secret keys of the node \( l \) (with their types): \( pk(l) : T_{pk} \), and \( sk(l) : T_{sk} \).

In order to model digital signatures and verification, I use the binary function symbols \( \text{sign} \) and \( \text{checksign} \), with the following equations:

\[
\text{sign}(t_{msg}, sk(l)).
\]

\[
\text{checksign}(\text{sign}(t_{msg}, sk(l)), pk(l)) = t_{msg}.
\]

note that in \( \text{checksign} \) the secret key, \( sk(l) \), and public key, \( pk(l) \), should match (i.e., they correspond to the same node ID), otherwise, the process gets stuck.

One-way hash function is defined as an unary function symbol \( h \), and no equation is defined for it. The absence of an inverse (equation) for \( h \) models the one-way property of \( h \). The assumption that \( h(t_{msg}) = h(t_{msg}) \) holds only when \( t_{msg} = t_{msg} \) ensures that \( h \) is collision-free. With these functions, the Ariadne protocol can be specified in the \( sr \)-calculus as follows:

\[
E_{ariadne} \equiv [P_{ariadne}]^{\sigma_{src}} | P_{int} | \prod_{l_{src}} | \prod_{l_{int}}^{\sigma_{int}} | [P_{dst}]^{\sigma_{dst}}.
\]

\( P_{src} \) defines the behavior of the source node: First, the source composes the header of the initial request, then, it computes a MAC on the header part, using the shared key with the destination. Afterwards, the source node broadcasts the initial request, and waiting for the corresponding reply.

\[
P_{ariadne} \equiv \text{let head}^{req} = (rreq, l_{src}, l_{dst}, ID) \text{ in}
\]

\[
\text{let MAC}_{ad} = \text{mac}(\text{head}^{req}, k(l_{src}, l_{dst})) \text{ in}
\]

\[
\langle \text{head}^{req}, \text{MAC}_{ad}, [], [] \rangle || \text{WaitRep}_{src}.
\]

When the source node receives a reply, it examines whether the address is \( l_{src} \), this is defined by the first construct = \( l_{src} \). This construct is a shorthand of the process \( x_{addressec} \) \( \ldots \) \( x_{addressec} = l_{src} \), and is borrowed from the specification language of the ProVerif tool. Similarly the IDs of the source and the destination, as well as the message type, \( rrep \), are also checked. If all of these verification steps succeed, in the next line, the source examines if the first ID in the node ID list belongs to its neighbor. If so, the source re-compute the initial MAC, and verify the signature.
located in the last place of the reply. This is expected to be signed by the destination. Afterwards,
in the source continually re-computes the all the per-hop hashes and signatures and compares them
with the corresponding received values. The conjunction \[ \bigwedge_{j \in \{1, \ldots, \text{last}\}} \ldots \] says that the three
lines within the brackets is repeated last number of times, where last is the length of the ID list.
Moreover, it also requires that the equality condition have to be valid in case of every \( j \). Finally,
in case every signature is correct the term \( \text{accept}(x_{\text{idList}}) \) is output to signal the acceptance of the
ID list \( x_{\text{idList}} \). The function \( j(x_{\text{idList}}) \) returns the \( j \)-th element of the list \( x_{\text{idList}} \).

\[
\text{WaitRep}_{\text{src}} \overset{\text{def}}{=} (= l_{\text{src}}, = \text{rrep}, = l_{\text{src}}, = l_{\text{dst}}, x_{\text{idList}}, x_{\text{sigList}}, x_{\text{sigDst}}).
\]

\[
\text{let head}^{\text{rrep}} = (\text{rrep}, l_{\text{src}}, l_{\text{dst}}) \text{ in}
\]

\[
\text{let hash}_0 = \text{mac}(\text{head}^{\text{rrep}}, k(l_{\text{src}}, l_{\text{dst}})) \text{ in}
\]

\[
[\{\text{head}^{\text{rrep}}, x_{\text{idList}}, x_{\text{sigList}} = \text{checksign}(x_{\text{sigDst}}, pk(l_{\text{dst}}))\}]
\]

\[
\bigwedge_{j \in \{1, \ldots, \text{last}\}} \{ \text{let hash}_j = h(l_j(x_{\text{idList}}), hash_{j-1}) \} \text{ in}
\]

\[
\text{let list}_j = [1(x_{\text{idList}}), \ldots, j(x_{\text{idList}})] \text{ in}
\]

\[
[\{\text{head}^{\text{rrep}}, \text{list}_j, \text{hash}_j = \text{checksign}(j(x_{\text{sigList}}), pk(l_j(x_{\text{idList}})))\}]
\]

\[
\langle \text{accept}(x_{\text{idList}}) \rangle.
\]

\( p_{\text{ariadne}} \) \( _{\text{int}} \) specifies the (uniform) behavior of the intermediate nodes, in particular, the index \( i \)
refers to the node \( \text{int}_i \). When a message is received for the first time, \( \text{int}_i \) checks whether its type is
\text{rreq}, and it examines if the last ID in the list belongs to its neighbor. Thereafter, \( \text{int}_i \) computes a hash and the signature on the received request, appends them to the request and re-broadcasts it.

\[
\begin{align*}
\text{WaitRep}^i_{\text{int}} & \overset{\text{def}}{=} (= \text{rreq}, x_{\text{src}}, x_{\text{dst}}, x_{\text{id}}, x_{\text{hashchain}}, x_{\text{idList}}, x_{\text{sigList}}). \\
& [\text{last}(x_{\text{idList}}) \in \sigma^i_{\text{int}}] \\
& \text{let head}^{\text{rreq}} = (\text{rreq}, x_{\text{src}}, x_{\text{dst}}, x_{\text{id}}) \text{ in} \\
& \text{let hash}_i = h(l_i(\text{hashchain})) \text{ in} \\
& \text{let sig}_i = \text{sign}(\text{head}^{\text{rreq}}, \text{hash}_i, [x_{\text{idList}}, l_i], x_{\text{sigList}}, sk(l_i)) \text{ in} \\
& \langle\langle\text{head}^{\text{rreq}}, \text{hash}_i, [x_{\text{idList}}, l_i], [x_{\text{sigList}}, \text{sig}_i]\rangle!\text{WaitRep}^i_{\text{int}}\rangle.
\end{align*}
\]

When \( \text{int}_i \) receives a message during waiting for a reply, the addressee and the message type
are verified, and followed by examining the next and the previous IDs in the list belong to the
neighbors of \( \text{int}_i \). If so, the reply is forwarded to the previous node.

\[
\begin{align*}
\text{WaitRep}^i_{\text{int}} & \overset{\text{def}}{=} (= l_i, = \text{rrep}, x_{\text{src}}, x_{\text{dst}}, x_{\text{idList}}, x_{\text{sigList}}, x_{\text{sigDst}}). \\
& [\text{prev}(x_{\text{idList}}, l_i) \in \sigma^i_{\text{int}}] [\text{next}(x_{\text{idList}}, l_i) \in \sigma^i_{\text{int}}] \\
& \langle\langle\text{prev}(x_{\text{idList}}, l_i), \text{rrep}, x_{\text{src}}, x_{\text{dst}}, x_{\text{idList}}, x_{\text{sigList}}, x_{\text{sigDst}}\rangle\rangle.
\end{align*}
\]

Note that the function \( \text{prev}(l_{\text{int}}, T_{\text{int}}, l_i; T_{\text{id}}) \) and \( \text{next}(l_{\text{int}}, T_{\text{int}}, l_i; T_{\text{id}}) \) return \text{undef} if \( l_i \) is not
in the list, otherwise, they return the element before and after \( l_i \), respectively. In case \( l_i \) is the
last element of \( l_{\text{int}} \), \( \text{next}(l_{\text{int}}, T_{\text{int}}, l_i; T_{\text{id}}) \) returns \( l_{\text{dst}} \), while if \( l_i \) is the first element of \( l_{\text{int}} \), then
\( \text{prev}(l_{\text{int}}, T_{\text{int}}, l_i; T_{\text{id}}) \) returns \( l_{\text{src}} \).

\( P_{\text{dst}} \) describes the behavior of the destination node. Within a session, when \( l_{\text{dst}} \) receives the
first message it examines the type of a the message, and the ID of the destination. After that, \( l_{\text{dst}} \)
computes the initial MAC, along with all the per-hop hashes and signatures, and compares them
with the corresponding received elements. If successful, then \( l_{\text{dst}} \) computes the signature on the
whole packet and sends back the reply to the last node in the list, \( \text{last}(x_{\text{idList}}) \).
2. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WIRELESS AD-HOC ROUTING PROTOCOLS

\[ P_{\text{ariadne}} \overset{def}{=} (rreq, x_{src}, l_{dst}, x_{id}, x_{hashchain}, x_{idList}, x_{sigList}), \]

\[ \begin{array}{l}
    (\text{last}(x_{idList}) \in \sigma_{\text{dst}}) \\
    \text{let } \text{head}^{\text{eq}} = (rreq, x_{src}, l_{dst}, x_{id}) \text{ in} \\
    \text{let } \text{hash}_0 = \text{mac}(\text{head}^{\text{eq}}, k(x_{src}, l_{dst})) \text{ in} \\
    \bigwedge j \in \{1, \ldots, \text{last}(x_{idList})\} \ (\text{let } \text{hash}_j = h(j(x_{idList}), \text{hash}_j-1) \text{ in} \\
    \text{let } \text{list}_j = [1(x_{idList}), \ldots, j(x_{idList})] \text{ in} \\
    [\text{head}^{\text{rep}}, \text{list}_j, \text{hash}_j] = \text{checksign}(j(x_{sigList}), pk(l_j(x_{sigList}))) \}
\end{array} \]

\[ \begin{array}{l}
    \text{let } \text{head}^{\text{rep}} = (rrep, x_{src}, l_{dst}) \text{ in} \\
    \text{let } \text{sig}_{\text{dst}} = \text{sign}((\text{head}^{\text{rep}}, x_{hashchain}, x_{idList}, x_{sigList})) \text{ in} \\
    \langle (\text{last}(x_{idList}), \text{head}^{\text{rep}}, x_{idList}, x_{sigList}, \text{sig}_{\text{dst}}) \rangle.
\end{array} \]

I will show that Definition 8 is violated with \( t_{\text{list}}, t_{\text{list}} = [l_{\text{int}}, l_{\text{att}}] \), where \( l_{\text{int}} \) and \( l_{\text{att}} \) are the IDs of a honest intermediate node and an attacker node, respectively. At the beginning, the attack topology \( T_{\text{top}} \) is empty, while the set of the edges in the route \( t_{\text{list}}, t_{\text{invalid}} \), is \( \{l_{\text{src}} - l_{\text{int}}, l_{\text{int}} - l_{\text{att}} \} \). Specifically, the reply received by the source has the form:

\[ \text{rep}/\text{repA} = (l_{\text{src}}, rrep, l_{\text{src}}, l_{\text{dst}}, ID, [l_{\text{int}}, l_{\text{att}}], [\text{sig}_{\text{int}}, \text{sig}_{\text{att}}], \text{sig}_{\text{dst}}). \]

Since the source accepts this reply, \( l_{\text{int}} \) must be a neighbor of \( l_{\text{src}} \), which can happen in two cases: The reply is sent by \( l_{\text{int}} \) or by the attacker \( l_{\text{att}} \). I examine the two cases in details, following the steps of the BDSR-algorithm:

1. Assuming one attacker node, in the first case, when \( l_{\text{src}} \) receives the reply \( \text{rep} \) from \( l_{\text{int}} \), we have the following message exchanges: The first state (State-1 in Figure 2.11) describes \( E_{\text{accept}}(t_{\text{list}}) \), in which \( \text{accept}(t_{\text{list}}) \) has been broadcast. The source broadcasts \( \text{accept} \) only when \( l_{\text{int}} \) is its neighbor. This backward deduction step is described by the labeled transitions trace

\[ E_{\text{accept}}(t_{\text{list}}) \overset{\nu_{\text{acc}}(x_{src})}{\longrightarrow} E_{\text{srcsentaccept}} \overset{\tau}{\longrightarrow} E_{\text{srctochecknbr}}, \]

where

![Figure 2.11: Analyzing Ariadne: The backward deduction steps based on the first case. Note that in the figure, the signature and hash verification steps are not illustrated, which happen after nbrcheck and before a message is sent. In the sr-calculus, signature verifications is modeled by a series of silent transitions.](image-url)
At the beginning, \( l_{\text{src}} \) does not have any neighbor, and \( \text{InitKnowlOfAtt} \) is the substitutions that represent the initial knowledge of the attackers. The state \( E_{\text{srcsentaccept}} \) is the same as \( E_{\text{accept}(t_{\text{int}})} \) except that the source node is \( \langle (\text{accept}(t_{\text{int}})) | P_{\text{src}} \rangle_{l_{\text{src}}}^\emptyset \). The series of silent transitions represent the verification of per-hop signatures performed by \( l_{\text{src}} \) and this trace ends with the neighbor check. In \( E_{\text{srcnbcheck}} \) the source has just performed a neighbor check, that is, the source process is \( \langle \text{SigVerifs} (\text{accept}(t_{\text{int}})) | P_{\text{src}} \rangle_{l_{\text{src}}}^\emptyset \), where \( \text{SigVerifs} \) is the code part right after the neighbor check. At this point, the neighborhood of \( l_{\text{src}} \) is updated with \( l_{\text{int}} \), because the term accept can only output at the end when the neighbor check succeeds. Finally, in \( E_{\text{srcnbchecknb}} \) the source is about to check if the first ID belongs to its neighbor, namely: \( \langle \text{accept}(t_{\text{int}}) | P_{\text{src}} \rangle_{l_{\text{src}}}^{\text{int}} \). The last (rightmost) silent transition in the trace corresponds to State-2 in Figure 2.11.

The next backward deduction steps, illustrated by State-3 and State-4 in Figure 2.11, represent the states in which the source receives and the intermediate node sends the reply \( \text{rep} \), respectively:

\[
l_{\text{int}} \rightarrow l_{\text{src}} : \text{rep} = (l_{\text{src}}, r\text{ep}, l_{\text{src}}, d\text{st}, \text{ID}, [l_{\text{int}}, l_{\text{att}}], \sigma_{\text{int}}, \sigma_{\text{att}}, \sigma_{\text{dst}}).
\]

Based on the transition system of \( sr \)-calculus, this is specified by the trace

\[
E_{\text{srcsentrep}}^{\text{rrep}(\text{rep})_{\sigma_{\text{int}}}} \quad \quad E_{\text{intsentrep}}^{\nu_{\text{int}}(l_{\text{src}}, l_{\text{int}})} \quad \quad E_{\text{rattsentrep}}^{\nu_{\text{att}}(\text{attsentrep})}.
\]

At the beginning, \( l_{\text{int}} \) does not have a neighbor (i.e., \( \sigma_{\text{int}} = \emptyset \)), however, \( l_{\text{src}} \) can only receive \( \text{rep} \) if it is the neighbor of \( l_{\text{int}} \). Hence, in \( E_{\text{intsentrep}} \), \( \sigma_{\text{int}} \) is updated to \( \{l_{\text{src}}\} \), which means that there is a bi-directional link between \( l_{\text{src}} \) and \( l_{\text{int}} \). State-5 says that before sending \( \text{rep} \), and performing signature verifications, \( l_{\text{int}} \) performed a neighbor check. At this point, we reason about how \( l_{\text{int}} \) could send \( \text{rep} \), namely, which message (and from whom) should \( l_{\text{int}} \) receive that message. Based on the list \( [l_{\text{int}}, l_{\text{att}}] \) in \( \text{rep} \), \( l_{\text{int}} \) has to receive a reply form \( l_{\text{att}} \). State-6 and State-7 represents the state \( E_{\text{intrcvrepA}} \) and \( E_{\text{attsentrepA}} \), in which \( l_{\text{int}} \) has just received \( \text{repA} \), and \( l_{\text{att}} \) has just sent it, respectively:

\[
l_{\text{att}} \rightarrow l_{\text{int}} : \text{repA} = (l_{\text{att}}, r\text{rep}, l_{\text{src}}, d\text{st}, \text{ID}, [l_{\text{int}}, l_{\text{att}}], \sigma_{\text{int}}, \sigma_{\text{att}}, \sigma_{\text{dst}}).
\]

Because \( l_{\text{int}} \) performed the neighbor check based on \( [l_{\text{int}}, l_{\text{att}}] \), there must be a bi-directional link between \( l_{\text{int}} \) and \( l_{\text{att}} \). The first six states belong to the honest phase \( \text{Ph-H1} \) in Figure 2.10. In State-7 we reason about the attacker’s behavior, namely, we get into phase \( \text{Ph-A} \).

In the rest part of this dissertation, I will explain the backward deduction informally to make it more easier to read. The formal interpretation of the deduction, based on labeled transition traces can be given in the same way as the transitions I provided above. Following the BDSR algorithm, first we examine if what would happen when the attacker has forwarded the received reply, correctly, according to the protocol. Based on the content of \( \text{repA} \), it follows that a reply (denoted by \( \text{repdst} \)) must have been sent directly by the destination \( l_{\text{att}} \), where:

\[
l_{\text{dst}} \rightarrow l_{\text{att}} : \text{repdst} = (l_{\text{att}}, r\text{rep}, l_{\text{src}}, d\text{st}, \text{ID}, [l_{\text{int}}, l_{\text{att}}], \sigma_{\text{int}}, \sigma_{\text{att}}, \sigma_{\text{dst}}).
\]

To achieve that \( l_{\text{dst}} \) will send this reply, \( l_{\text{dst}} \) must have received the request, \( \text{reqA} \), from \( l_{\text{att}} \):

\[
l_{\text{att}} \rightarrow l_{\text{dst}} : \text{reqA} = (r\text{req}, l_{\text{src}}, d\text{st}, \text{ID}, \text{hashatt}, [l_{\text{int}}, l_{\text{att}}], \sigma_{\text{int}}, \sigma_{\text{att}}).
\]
However, the last two messages means that there is a bi-directional link between \( l_{att} \) and \( l_{dst} \). At this point we have \( T_{invalid} \subseteq T_{top} \), hence, from this point, the ideal system can always simulate the deduction of the real system. Consequently, this deduction branch cannot lead to an attack because the route defined by \( l_{list} \) is a valid route from now on.

We return to the beginning of phase \( Ph-A \) and follow the points (I-IV.) of the BDSR algorithm. We examine how the attacker could compose each part of the reply message \( rep.A \). Recall that to be successful the attacker must obtain all the parts of \( rep.A \). According to the algorithm, we start with the term that has the highest priority and weight, which is the signature \( sig_{dst} \) computed by \( l_{dst} \). The attacker cannot compute \( sig_{dst} \) because it does not posses the private key \( sk(l_{dst}) \), and we assumed that private keys will not be leaked during the route discovery process. Therefore, \( l_{att} \) can only obtain \( sig_{dst} \) if it receives a reply that contains \( sig_{dst} \). Because \( sig_{dst} \) is computed on the list \([l_{int}, l_{att}]\), it must be in the the reply \( rep.dst \), sent by \( l_{dst} \) to \( l_{att} \) (States-8-9). This bring us back to the situation similar to the previous case, namely, \( T_{invalid} \subseteq T_{top} \) becomes valid after State-11. To summarize, the first case cannot result in an attack scenario.

![Diagram of the BDSR algorithm](image_url)

Figure 2.12: Analyzing Ariadne with BDSR: The backward deduction steps based on the second case.

2. In the second case (Figure 2.12), the source accepts the reply \( rep.A \) sent by the attacker \( l_{att} \). States 1-2 are interpreted like in the first case. As State-3 shows, \( l_{src} \) received the reply from \( l_{att} \):

\[
l_{att} \rightarrow l_{src} : (l_{src}, rrep, l_{src}, l_{dst}, ID, [l_{int}, l_{att}], [sig_{int}, sig_{att}], sig_{dst}).
\]

In State-2 and State-3 \( T_{top} \) is updated with two uni-directional links, \( l_{src} \rightarrow l_{int} \) and \( l_{att} \rightarrow l_{src} \). At this point we get into the attacker phase \( Ph-A \). I skip discussing the case when \( l_{att} \) forwards correctly the reply which it received, because the or the similar reason as the first case, it cannot lead to an attack scenario. I examine how the attacker could compose each part of \( rep.A \) (steps I-IV of BDSR), and start with the highest priority/weight element, \( sig_{dst} \).

Like in the first case, the reply (denoted by \( rep.dst \)) must have been sent by the destination, \( l_{dst} \) to \( l_{att} \) (shown in State-5):

\[
l_{dst} \rightarrow l_{att} : rep.dst = (l_{att}, rrep, l_{src}, l_{dst}, ID, [l_{int}, l_{att}], [sig_{int}, sig_{att}], sig_{dst}).
\]
At this point, we reach the attacker node and step into the phase $Ph-A$ again. The deduction cannot lead to an attack scenario when the attacker forwards $reqA$ correctly, because this would mean that $l_{att}$ must have received the request

$$l_{int} \rightarrow l_{att} : (rreq, l_{src}, l_{dst}, ID, hash_{att}, [[l_{int}], [sig_{int}])$$

from $l_{int}$, and before this, $l_{int}$ must have received the initial request from $l_{src}$. This results in $T_{invalid} \subseteq T_{att}$. Therefore, we examine how the attacker could obtain each part of $reqA$. Specifically, we examine how node $l_{att}$ could obtain $hash_{att}$, $sig_{int}$ and $sig_{att}$.

First, to compute $sig_{att}$ where

$$sig_{att} = sign((rreq, l_{src}, l_{dst}, ID, hash_{att}, [[l_{int}], [sig_{int}]), sk(l_{att}))$$

the attacker has to obtain $sig_{int}$ and $hash_{att}$. How the signature $sig_{int}$, computed by node $l_{int}$, can be obtained? Can any request message contains $sig_{int}$, which will be received or intercepted by $l_{att}$? The attacker tries to append some new node ID, $l_{new}$, which has not appeared during the backward deduction, to the list $[l_{int}]$, and getting $[l_{int}, l_{new}]$. Then, we examine whether $sig_{int}$ can be found in the request, $req_{new}$, which contains the list $[l_{int}, l_{new}]$:

$$req_{new} = (rreq, l_{src}, l_{dst}, ID, hash_{new}, [[l_{int}, l_{new}], [sig_{int}, sig_{new}]).$$

The answer is yes, and this message should be sent from $l_{new}$ to $l_{att}$ (in States-10-11, $req_{new}$ is received by $l_{att}$, and it is sent by $l_{new}$, respectively). $T_{top}$ is updated with the link $l_{new} \rightarrow l_{att}$. Before this, $l_{new}$ should obtain the request

$$req_{int} = (rreq, l_{src}, l_{dst}, ID, hash_{int}, [[l_{int}], [sig_{int}]),$$

from $l_{int}$ (States-12-13-14), who must have received the initial request output by $l_{src}$ (States-15-16-17). The topology $T_{top}$ is updated with two bi-directional links $l_{new} \rightarrow l_{int}$, and $l_{int} \rightarrow l_{src}$.

Besides $sig_{int}$, the attacker also has to compute $hash_{att}$. To compute $hash_{att}$, $hash_{att} = h((l_{att}, MAC_{sd}))$, the attacker has to obtain $MAC_{sd}$, which is the MAC computed by the source on the initial request using the key it shares with the destination. The question is that can $MAC_{sd}$ be a part of an request/reply message, which can be obtained by $l_{att}$? The answer is yes, because the initial request sent by $l_{src}$ contains $MAC_{sd}$ (illustrated in State-18-19). The link $l_{src} \rightarrow l_{att}$ is added to $T_{top}$.

With these deduction branches it follows that the attacker $l_{att}$ can obtain or compute all the parts of $reqA$. Formally, on every branch, we could get back to the state $E_{reqlint}$, and $T_{invalid} \not\subseteq T_{top}$ is valid throughout the deduction procedure. Based on this deduction path, the Definition 8 is violated because the labeled transition traces that correspond to this backward deduction, is allowed in the real system, however, this trace cannot be simulated in the ideal system. The reason is that in the ideal system, the last labeled transition step ($last-TRANS$), is not allowed in the last deduction branch, while it is allowed in the real system. Hence, the ideal system cannot simulate this transition of the real system.

To summarize the analysis, I proved that the Ariadne protocol is insecure when one attacker node is assumed in the network.
2.7.4 Analyzing the security of endairA based on the BDSR algorithm

In this subsection, I analyze the security of the endairA protocol. I start with the specification of the processes in the sr-calculus:

\[ E_{\text{endairA}} \overset{\text{def}}{=} !P^\text{src}_{\text{endairA}} \sigma_{\text{src}} | \prod_{i \in 1, \ldots, n} !P^\text{int}_{\text{endairA}} \sigma_{\text{int}} | !P^\text{dst}_{\text{endairA}} \sigma_{\text{dst}}. \]

\[ P^\text{src}_{\text{endairA}} \overset{\text{def}}{=} \nu ID. \langle (\text{rreq}, l_{\text{src}}, d_{\text{dst}}, [\cdot, [\cdot]]) \rangle \text{!WaitRep}_{\text{src}}. \]

\[ \text{WaitRep}_{\text{src}} \overset{\text{def}}{=} \]

\[ \ni \text{let } \text{head}^{\text{rep}} = (\text{rrep}, l_{\text{src}}, d_{\text{dst}}) \text{ in}
\begin{align*}
&\langle \text{checksign}(x_{\text{sigDst}}(d_{\text{dst}}), \text{pk}(l_{\text{dst}})) \rangle
\end{align*}
\]

\[ \text{let } \text{sig}^{\text{ListTill}_0} = [x_{\text{sigDst}}] \text{ in}
\begin{align*}
&\text{let } \text{tillsig}_0 = (\text{head}^{\text{rep}}, x_{\text{idList}}, \text{sigListTill}_0) \text{ in}
\end{align*}
\]

\[ \text{let } j \in \{1, \ldots, \text{last} \} \text{ in}
\begin{align*}
&\langle \text{checksign}(j(x_{\text{sigList}}), \text{pk}(j(x_{\text{idList}}))) \rangle
\end{align*}
\]

\[ \text{let } \text{sig}_j = \text{sign}((\text{tillsig}_j-1), \text{sk}(j(x_{\text{idList}}))) \text{ in}
\begin{align*}
&\text{let } \text{sigListTill}_j = [\text{sigListTill}_j-1, \text{sig}_j] \text{ in}
\end{align*}
\]

\[ \text{let } \text{tillsig}_j = (\text{head}^{\text{rep}}, x_{\text{idList}}, \text{sigListTill}_j) \text{ in}
\]

\[ \langle \text{accept}(x_{\text{idList}}) \rangle. \]

\[ P^\text{int}_{\text{endairA}} \overset{\text{def}}{=} \langle \text{rreq}, x_{\text{src}}, d_{\text{dst}}, x_{\text{id}}, x_{\text{idList}} \rangle.
\]

\[ \langle \text{prev}(x_{\text{idList}}, l_i) \in \sigma_{\text{int}} \rangle \langle \text{accept}(x_{\text{src}}, x_{\text{dst}}, x_{\text{id}}, x_{\text{idList}}, l_i) \rangle \text{!WaitRep}_{\text{int}}. \]

\[ \text{WaitRep}_{\text{int}} \overset{\text{def}}{=} \]

\[ \ni \text{let } \text{head}^{\text{rep}} = (\text{rrep}, l_{\text{src}}, d_{\text{dst}}) \text{ in}
\begin{align*}
&\langle \text{checksign}(x_{\text{sigDst}}(d_{\text{dst}}), \text{pk}(l_{\text{dst}})) \rangle
\end{align*}
\]

\[ \text{let } \text{sig}^{\text{ListTill}_0} = [x_{\text{sigDst}}] \text{ in}
\begin{align*}
&\text{let } \text{tillsig}_0 = (\text{head}^{\text{rep}}, x_{\text{idList}}, \text{sigListTill}_0) \text{ in}
\end{align*}
\]

\[ \text{let } j \in \{1, \ldots, \text{last} \} \text{ in}
\begin{align*}
&\langle \text{checksign}(j(x_{\text{sigList}}), \text{pk}(j(x_{\text{idList}}))) \rangle
\end{align*}
\]

\[ \text{let } \text{sig}_j = \text{sign}((\text{tillsig}_j-1), \text{sk}(j(x_{\text{idList}}))) \text{ in}
\begin{align*}
&\text{let } \text{sigListTill}_j = [\text{sigListTill}_j-1, \text{sig}_j] \text{ in}
\end{align*}
\]

\[ \text{let } \text{tillsig}_j = (\text{head}^{\text{rep}}, x_{\text{idList}}, \text{sigListTill}_j) \text{ in}
\]

\[ \langle \langle \text{prev}(x_{\text{idList}}, l_i), \text{rrep}, x_{\text{src}}, d_{\text{dst}}, x_{\text{idList}}, x_{\text{sigDst}}, x_{\text{sigList}} \rangle \rangle \rangle. \]

\[ P^\text{dst}_{\text{endairA}} \overset{\text{def}}{=} \langle \text{rreq}, x_{\text{src}}, d_{\text{dst}}, x_{\text{id}}, x_{\text{idList}} \rangle.
\]

\[ \langle \text{last}(x_{\text{idList}}) \in \sigma_{\text{dst}} \rangle \text{!WaitRep}_{\text{dst}}. \]

Again, I make the analysis easy to read by discussing the the deduction steps informally. The corresponding labeled transitions, and extended networks can be given in the same way as in the analysis of the Ariadne protocol. I distinguish the next settings before performing a systematic analysis based on the BDSR algorithm.
1. **Assuming only one attacker node in the network:**

   **Case I.** The list $l_{list}$ includes $l_{att}$ as the first element, namely, $l_{list} = [l_{att}, l_{int}^1, \ldots, l_{int}^k]$ for some $k$. In this case, the reply, denoted by $repA$, which is received and accepted by $l_{src}$, must be sent by the attacker $l_{att}$ (otherwise, the first ID in $l_{list}$ would belong to a honest node).

   $$ l_{att} \rightarrow l_{src} : repA = (l_{src}, rrep, l_{src}, l_{dst}, ID, [l_{att}, l_{int}^1, \ldots, l_{int}^k], sig_{dst}, [sig_{int}^k, \ldots, sig_{int}^1, sig_{att}]). $$

   We step into phase Ph-A and reason about how $l_{att}$ could compose $repA$. If $l_{att}$ forwards the reply accordingly to the protocol, the deduction will not lead to an attack scenario because $T_{invalid} \subseteq T_{top}$ will hold. Namely, this will be resulted after the following backward deduction steps

   $$ l_{att} \overset{rep^1}{\leftarrow} l_{int}^1 \overset{rep^2}{\leftarrow} \ldots \overset{rep^k}{\leftarrow} l_{int}^k \overset{rep^d}{\leftarrow} l_{dst}. $$

   We examine how the attacker can obtain the highest priority/weight part in $repA$, which is $sig_{int}^1$, the signature of node $l_{int}^k$. The attacker cannot compute this signature because it does not have the signing key $sk(l_{int}^k)$. Hence, $l_{att}$ can only obtain $sig_{int}^1$ by receiving/intercepting a request/reply which contains $sig_{int}^1$. In the following, we reason about which request/reply message can contain $sig_{int}^1:$

   $$ sig_{int}^1 = sign((rrep, l_{src}, l_{dst}, ID, [l_{att}, l_{int}^1, \ldots, l_{int}^k], sig_{dst}, [sig_{int}^k, \ldots, sig_{int}^1, sig_{att}], sk(l_{int}^k)). $$

   The important difference between endairA and Ariadne is that in endairA, signatures are computed on the whole list $l_{list}$ Therefore, the reply messages that contain node ID lists differ from $l_{list}$ cannot include signatures in $repA$. There are two cases:

   **C1:** $sig_{int}^1$ can be included in $repA$. However, as already discussed in the previous point, $repA$ must traverse on the path $l_{list}$, which results in $T_{invalid} \subseteq T_{top}$.

   **C2:** $sig_{int}^1$ can be included in a request or a reply that contains $sig_{int}^1$ in the place of the session ID. This is feasible because in that place we expect any data with type string ($T_{str}$), which involves the type of signatures ($T_{sign}$). Hence, we examine how the attacker $l_{att}$ could obtain the request

   $$ req' = (rreq, l_{src}, l_{dst}, sig_{int}^1, l_{list}). $$

   This request cannot be sent directly by the source, because based on the protocol, the initial request only allows a data with session ID type ($T_{sid}$) at the fourth place, which cannot be a signature. Hence, $req'$ must have been sent on the path of one or more intermediate nodes

   $$ l_{att} \overset{req^{(1)}}{\leftarrow} l_{int}^{(1)} \overset{req^{(2)}}{\leftarrow} \ldots \overset{req^{(m)}}{\leftarrow} l_{int}^{(m)} \overset{req^{(m+1)}}{\leftarrow} l_{att}. $$

   However, since the initial request sent by $l_{src}$ does not allow a signature to be in the place of the session ID, $l_{int}^{(m)}$ must receive a request from the attacker node $l_{att}$. Due to the assumption of one attacker node, node $l_{att}$ and $l_{att}$ must be the same. At this point we get into a loop and stop (because again, we try find out how $sig_{int}^1$ can be obtained).

   **Case II.** $l_{att}$ is not the first element of the list $l_{list}$, namely, $l_{list} = [l_1, \ldots, l_{att}, \ldots]$. In this case the backward deduction will reach the point where $T_{invalid} \subseteq T_{top}$ holds, hence, no attack scenario will be resulted.

   To summarize the analysis: I proved that the endairA protocol is secure if one attacker node is assumed.
2. There can be more than one attacker node: In this case we continue case C2 from the point where we get into a loop. This time, \( l'_{\text{att}} \) and \( l_{\text{att}} \) can be two different attacker nodes. We analyze how (i.e., on which path) the second attacker node \( l'_{\text{att}} \) can obtain \( \text{sig}_{\text{int}} \). The second attacker node \( l'_{\text{att}} \) must receive the reply sent by the honest node \( l_{\text{int}} \), because only this reply contains \( \text{sig}_{\text{int}} \). There can be two possibilities:

- We have to extend the route accepted by the source \( (t_{\text{list}}) \) with \( l'_{\text{att}} \) by inserting it before \( l_{\text{int}} \). Namely, \( t_{\text{list}} = [l_{\text{att}}, l'_{\text{att}}, l_{\text{int}}, \ldots, l_{\text{att}}] \). Node \( l'_{\text{att}} \) receives the following reply, denoted by \( \text{rep}1 \), from \( l_{\text{int}} \):

\[
l_{\text{int}} \rightarrow l'_{\text{att}} : \text{rep}1 = (l'_{\text{att}}, \text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [l_{\text{att}}, l'_{\text{att}}, l_{\text{int}}, \ldots, l_{\text{att}}], \text{sig}_{\text{list}}, [\text{sig}_{\text{int}}, \ldots])
\]

Then, we keep track backward the possible path on which this reply has traversed, and found that from \( l_{\text{int}} \) backward to \( l_{\text{dst}} \) the reply must be forwarded correctly according to the protocol:

\[
l_{\text{int}} \xrightarrow{\text{rep}k} \cdots \xrightarrow{\text{req}1} l_{\text{int}} \xrightarrow{\text{req}k} l_{\text{dst}},
\]

and the corresponding requests must be forwarded correctly from \( l_{\text{int}} \) to \( l_{\text{dst}} \)

\[
l_{\text{int}} \xrightarrow{\text{req}k} \cdots \xrightarrow{\text{rep}1} l_{\text{int}} \xrightarrow{\text{rep}k} l_{\text{dst}},
\]

where \( \text{req}k = (\text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [l_{\text{att}}, l'_{\text{att}}, l_{\text{int}}, \ldots, l_{\text{att}}]), \ldots \), \( \text{req}1 = (\text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [l_{\text{att}}, l'_{\text{att}}, l_{\text{int}}]) \). Further, \( l'_{\text{att}} \) must have sent the request, \( \text{req}A' \), \( \text{req}A' = (\text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [l_{\text{att}}, l'_{\text{att}}]) \). At this point we get into the attacker phase \( \text{Ph-A} \), and reason about how node \( l_{\text{att}} \) could compose \( \text{req}A' \). Since \( l'_{\text{att}} \) cannot generate the fresh session ID, it must be received as a part of the request sent by \( l_{\text{src}} \). Now, the main question is that (i.) does the second attacker, \( l'_{\text{att}} \), receive the request directly from the first attacker, \( l_{\text{att}} \), or (ii.) from an intermediate honest node \( l_{\text{int}} \). The first case means that there is a link between \( l_{\text{att}} \) and \( l'_{\text{att}} \), and because the reply was sent by \( l_{\text{att}} \) to \( l_{\text{src}} \), there is also a link between \( l_{\text{att}} \) and \( l_{\text{src}} \).

To summarize, this means that the route, \( [l_{\text{att}}, l'_{\text{att}}, l_{\text{int}}, \ldots, l_{\text{att}}] \), accepted by the source is valid, thus, no attack is detected. In the second case, we have the following messages:

\[
l_{\text{int}} \rightarrow l'_{\text{att}} : (\text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [l'_{\text{list}}, l_{\text{int}}]),
\]

for some list \( l'_{\text{list}} \) that begins with \( l_{\text{att}} \). We examine the case when \( l'_{\text{list}} = [l_{\text{att}}] \), that is, \( l_{\text{int}} \) must have received the request \( (\text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [l_{\text{att}}]) \), which must have been sent by \( l_{\text{att}} \). In order to send this request, \( l_{\text{att}} \) must receive the initial request, \( \text{reqinit} = (\text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [\cdots]) \) from \( l_{\text{src}} \).

To summarize, I detected the following attack scenario in case of two attacker nodes:

The following messages are sent during the attack scenario: As the result of the attack, the two attacker nodes \( l_{\text{att}} \) and \( l'_{\text{att}} \) can achieve that the source \( l_{\text{src}} \) accept the invalid route \( [l_{\text{att}}, l'_{\text{att}}, l_{\text{int}}, \ldots, l_{\text{att}}] \) instead of the valid \( [l_{\text{att}}, l_{\text{int}}, l'_{\text{att}}, l_{\text{int}}, \ldots, l_{\text{int}}] \) (shown in Figure 2.13).

1. \( \text{reqinit} = (\text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [\cdots]) \);
2. \( \text{req}A = (\text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [l_{\text{att}}]) \);
3. \( \text{req}0 = (\text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [l_{\text{int}}]) \);
4. \( \text{req}A' = (\text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [l_{\text{att}}, l'_{\text{att}}]) \);
5. \( \text{req}1 = (\text{req}, l_{\text{src}}, l_{\text{dst}}, ID, [l_{\text{att}}, l'_{\text{att}}, l_{\text{int}}]) \);
Figure 2.13: The figure illustrates the attack against the endairA protocol. The upper figure presents the request phase, while the one below it shows the reply phase of the attack scenario. Node IDs $l_{att}$ and $l'_{att}$ represent the two attacker nodes, $l_{src}$ and $l_{dst}$ are the IDs of the source and destination nodes, while the remaining IDs belong to the intermediate honest nodes.

In the request phase, after receiving the initial request, the attacker node $l_{att}$ appends its ID and broadcasts it, and so does the intermediate node $l^0_{int}$. The second attacker node $l'_{att}$ replaces $l^0_{int}$ with its own ID and broadcasts the modified request. The request is forwarded by the honest nodes $l^1_{int}, \ldots, l^k_{int}$, according to the protocol.

6. repdst = $(l^k_{int}, rrep, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^1_{int}, \ldots, l^k_{int}], \text{sig}_{dst})$;
7. repk = $(l^{k-1}_{int}, rrep, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^1_{int}, \ldots, l^k_{int}], \text{sig}_{dst}, [\text{sig}^k_{int}, \ldots, \text{sig}^2_{int}])$;
8. rep1 = $(l'_{att}, rrep, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^1_{int}, \ldots, l^k_{int}], \text{sig}_{dst}, [\text{sig}^k_{int}, \ldots, \text{sig}^1_{int}])$;
9. repA = $(l_{src}, rrep, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^1_{int}, \ldots, l^k_{int}], \text{sig}_{dst}, \text{sig}^k_{int}, \ldots, \text{sig}^1_{int}, \text{sig}'_{att}, \text{sig}_{att})$.

In the reply phase, the reply is sent back by $l_{dst}, l^1_{int}, \ldots, l^k_{int}$ according to the protocol. When $l'_{att}$ receives the reply $rep1$ from $l^1_{int}$, it should not send it to $l^0_{int}$, because $l^0_{int}$ will drop the reply since the ID $l^0_{int}$ does not appear in the ID list. Hence, $l'_{att}$ needs to find another way to forward information to $l_{att}$.

In this interleaving session the attacker $l_{att}$ receives a request from some honest node, and it replaces the session ID, $ID'$, with the signature $\text{sig}^1_{int}$. Then, the modified request is broadcast towards the first attacker $l_{att}$. Since the honest nodes forwards this request without changing the session ID, $l_{att}$ will receive $\text{sig}^1_{int}$ in message $reqj$, and uses it for constructing a “proof” for the incorrect route $t_{list}$ in the another route discovery session.

Interleaving1: $reqA'' = (rreq, l_{src}, l_{dst}, \text{sig}^1_{int}, [l'_{att}])$;
Interleaving2: $reqt = (rreq, l_{src}, l_{dst}, \text{sig}^1_{int}, [l'_{att}, l^1_{int}])$;


Interleavingm: $reqj = (rreq, l_{src}, l_{dst}, \text{sig}^1_{int}, [l'_{att}, l^1_{int}, \ldots, l^j_{int}])$;

In order to construct the proof for the invalid route $t_{list}$, $l_{att}$ has to obtain all of the signatures in $repA$. All of these signatures can be obtained similarly as $\text{sig}^1_{int}$, in different interleaving sessions.

39
- \( l'_{\text{att}} \) is not the part of \( l_{\text{int}} \), and only overhears the reply sent by \( l^1_{\text{int}} \). This case also results in an attack scenario shown in Figure:

![Figure 2.14: The figure illustrates another possible attack against the endairA protocol. In this scenario, \( l'_{\text{att}} \) is not the part of the invalid route \( (l_{\text{int}}) \) that has been accepted by \( l_{\text{src}} \), but it is neighbor of \( l^1_{\text{int}} \) and \( l_{\text{int}} \). In particular, the two attacker nodes can achieve that \( l_{\text{src}} \) accepts the route \([l_{\text{att}}, l^1_{\text{int}}, \ldots, l^k_{\text{int}}]\) instead of the correct route \([l_{\text{att}}, l^0_{\text{int}}, l^1_{\text{int}}, \ldots, l^k_{\text{int}}]\).

In this attack, the request messages are sent by \( l_{\text{att}} \) and the honest nodes in a correct way. The attacker \( l'_{\text{att}} \) stays idle in the request phase. In addition, the corresponding reply is sent back correctly by \( l^1_{\text{int}}, l^2_{\text{int}}, \ldots, l^k_{\text{int}} \). Instead of sending the correct reply to \( l_{\text{src}} \), the attacker \( l_{\text{att}} \) waits for the signatures \( \text{sig}_{\text{dst}}, \text{sig}_{l^1_{\text{int}}, \ldots, \text{sig}_{l^1_{\text{int}}} \), which can be sent by \( l'_{\text{att}} \) in interleaving sessions. In each session, \( l'_{\text{att}} \) broadcasts a request message that contains the signature(s) in place of the session ID.

To summarize the analysis: I proved that the endairA protocol is insecure if more than one attacker node is assumed, who can cooperate.

### 2.8 \( sr\text{-verif} \): On automating the verification

In this section, I present a novel automated verification technique for on-demand source routing protocols, based on the BDSR algorithm, and backward resolutions performed on logic rules. I named this proposed verification method as \( sr\text{-verif} \).

#### 2.8.1 The concept of the verification method

In this subsection, I discuss the concept of the proposed verification method. In addition, I give an overview of the main differences among the proposed \( sr\text{-verif} \) method, the method proposed in [Th05 , 2010], and the method used by the Proverif tool [13].

In the ProVerif verification tool [13] the input of the tool is the specification of security protocols in the syntax of the applied \( \pi\text{-calculus} \) (Figure 2.15). The main advantage of using a calculus as a specification language is that the operation of protocols can be unequivocally and precisely modeled in it. The tool then translates the protocol specification to logic rules for performing automatic verification.

My proposed technique was inspired by the method used in Proverif, however, as opposed to ProVerif it is designed for verifying routing protocols and includes numerous novelties such as the modeling of broadcast communications, neighborhood, transmission range, and considers an attacker model specific to wireless ad hoc networks instead of the Dolev-Yao attacker model.

One important difference between the modeling of secure routing protocols and ordinary security protocols is that while in security protocols each communicating entity can have different internal operation and structure, in case of secure routing protocols each communication entity usually has the same uniform structure: During the route discovery each node can be (i) source node, or (ii) intermediate node, or (iii) destination node. Hence when using the Proverif tool to
model secure routing protocols, in case of the network including \( n \) nodes, the user has to describe the operation of all \( n \) nodes despite the fact that they are the same up to renaming variables and names. In contrast, in my method the user is required to specify only the “general” operation of nodes, which represents any node.

Following this concept, I proposed the verification method in [Th05 , 2010]. As Figure 2.16 shows, the operation of routing protocols are specified in the syntax of processes of the simplified \( sr\)-calculus. This is then translated to Horn-clauses using translation rules. This set of clauses is called protocol rules. In addition, the topology and the initial knowledge of the attacker node are specified by a set of facts, while the computation ability of the attacker node is specified by the set of Horn-clauses. The clauses that specify the attacker computation ability are called attacker rules. The deductive algorithm is based on the resolution steps accomplished over these clauses and facts in a forward search manner. The concept of the improved method is shown in Figure 2.16.

The basic idea of my approach is that initially an invalid route \( r \), which is represented by an ID list \([\text{List}_{\text{invalid}}]\) of different IDs, is supposed to be accepted at the end of the route discovery. The task of the verification algorithm is to confirm this assumption by finding a sequence of message exchanges along with a topology in which both the destination and source accept \([\text{List}_{\text{invalid}}]\) as a valid route, or to give a refutation in case no attack scenario can be found. At the beginning, the topology includes only the attacker node, the source node and the destination node without any edges. This is iteratively updated with new edges and nodes during the verification. My method supports both uni and bi-directional edges. The verification consists in at most \( n \) rounds, where \( n \) is a bound parameter which represents the maximal number of honest node IDs in \([\text{List}_{\text{invalid}}]\). In each round, \( \binom{n}{c} + a \) number of ID lists are examined, where \( c \) ranges from 1 to \( n \), and \( a \) is the number of attacker IDs. In particular, an invalid ID list of \( c \) honest node IDs, \([l_{p1}, \ldots, l_{pc}]\), along with the additional lists created by inserting unique IDs of the attackers into every possible places in \([l_{p1}, \ldots, l_{pc}]\) are analyzed. Additionally, \( sr\)-verif also considers such \([\text{List}_{\text{invalid}}]\) that contains only the IDs of the attackers. Whenever an attack is detected, \( sr\)-verif returns the attack scenario and stops. During the searching procedure, \( sr\)-verif aims at finding an attack as short as possible,
however, due to the diversity of source routing protocols, in some cases the returned attack may be not the shortest.

One advantage of my approach is that by searching for attacks in case of different invalid ID lists we cover all the possibilities without redundancy. Note that the IDs \( l_1^p, \ldots, l_n^p \) are symbolic, they are distinguished only by names because initially they are not equipped with any further information such as location and neighborhood. Therefore, the order of \( l_1^p, \ldots, l_n^p \) in the list is disregarded. In addition, I emphasize that only the number of honest nodes in \([\text{List}_{\text{invalid}}]\) is upper bounded. On the other hand, the price I have to pay is that my method, in the general case, can only prove that a routing protocol is flawless within \( n \) rounds. Finally, I have applied my method to verify some representative routing protocols and found that attacks were detected within only a few steps. In case of the SRP protocol an attack is detected in the first round, whilst an attack scenario against the Ariadne protocol is returned in the second round.

### 2.8.2 Specifying on-demand source routing protocols

In this section, I give an overview of the formal language with which the operation of routing protocols can be specified (the leftmost box in Figure 2.16). Using directly the formal syntax of the sr-calculus for specifying routing protocols is a bit cumbersome, because it includes several complex notations. Hence, for automating the verification I introduce a variant of the sr-calculus with a simplified syntax, which can be edited in an easier way in text format. Terms are defined in the same way as in the sr-calculus. The syntax of the processes, which model the internal operation of nodes, are simplified in the following way:

\[
P, \ Q, \ R :::= \\
\quad \text{broadcast} \ (t) \ . \ P \quad \text{broadcast by source node} \\
\quad \text{broadcast} \ . \ P \quad \text{broadcast by other nodes} \\
\quad \text{unicast} \ ((y, t)) \ . \ P \quad \text{unicast by destination node} \\
\quad \text{unicast} \ . \ P \quad \text{unicast by other nodes} \\
\quad \text{receive} \ . \ P \quad \text{receive request} \\
\quad \text{receive reply} \ . \ P \quad \text{receive reply} \\
\quad \text{parallel composition} \ (Q, P) \quad \text{parallel composition} \\
\quad \text{restriction} \ (\text{new} \ n) \ . \ P \\
\quad \text{replication} \ !P \\
\quad \text{nil} \\
\quad \text{constructor application} \ (\text{let} \ (x = g(t_1, \ldots, t_n)) \ \text{in} \ P) \quad \text{let} \\
\quad \text{destructor application} \ \text{let} \ (x = t) \ \text{in} \ P \\
\quad \text{let or} \ (x = t_1) \ \text{or} \ (x = t_1) \ \text{in} \ P \\
\quad \text{let or} \ (x = t) \ \text{or} \ (x = t) \ \text{in} \ P \\
\quad \text{neighbor} \ \text{if} \ \text{nbr} \ (y, y) \ \text{then} \ P \\
\quad \text{accept} \ (y, \ \text{List}) \\
\]

The processes \((t) \ . \ P\) in the sr-calculus is replaced by \(\text{broadcast} \ (t) \ . \ P\) and \(\text{broadcast} \ . \ P\), respectively, in the simplified syntax. The receive action \((x) \ . \ P\) is replaced by two processes \(\text{recvmsg} \ (t) \ . \ P\) and \(\text{recvmsg} \ . \ P\). The processes \(\text{unicast} \ ((y, t)) \ . \ P\) and \(\text{unicast} \ . \ P\), respectively represent sending a message \(t\) to node with the ID \(y_{\text{nid}}\). The index \(\text{nid}\) refers to a variable that has the node ID type.

- Both processes \(\text{broadcast} \ (t) \ . \ P\) and \(\text{broadcast} \ . \ P\) first broadcast \(t\), which is followed by the running of process \(P\). The reason why I distinguish \(\text{broadcast}\) from \(\text{broadcast}\) is that they correspond to different logic rules in the automated deduction algorithm.

- Processes \(\text{unicast} \ ((y_{\text{nid}}, t)) \ . \ P\) and \(\text{unicast} \ . \ P\) first send \((y_{\text{nid}}, t)\) then are continued with \(P\). Node ID \(y_{\text{nid}}\) at the beginning of \((y_{\text{nid}}, t)\) specifies the addressee of the message \(t\). Note that every message sent in the network is overheard, however, honest nodes drop the message if they are not the addressee. \text{unicast} is used when a message is sent by the destination.

- Processes \(\text{recvmsg} \ (t) \ . \ P\) and \(\text{recvmsg} \ . \ P\) wait for the term request and reply \(t\), respectively. In case the received term is equal to \(t\), \(P\) will run, otherwise, it gets stuck and stays idle.
A composition $P|Q$ behaves as processes $P$ and $Q$ running in parallel. Each may interact with the other on channels known to both, or with the outside world, independently of the other.

A restriction (new n)$P$ is a process that makes a new, private name $n$, and then behaves as $P$.

A replication $!P$ is the parallel composition of infinite instances of $P$. This is used to model infinite parallel sessions of a protocol.

The nil process does nothing, used to model process termination.

Process letdst $(x = g(t_1, \ldots, t_n))$ in $P$ else $Q$ tries to evaluate $g(t_1, \ldots, t_n)$ if this succeeds, $x$ is bounded to the result and $P$ is executed, otherwise, $Q$ is executed. For instance, a typical destructor can be verification of digital signature as $\text{checksign}(\text{sign}(x, sk(y)), pk(sk(y)))$, where the constructor $pk(sk(y))$ represents the public key generated from the given secret key. In short, we can say that the destructor function is a kind of “inverse” function of constructor function: For example, if the constructor function is signature then destructor function is $\text{checksign}$, and if the constructor function is encryption then the corresponding destructor function is decryption.

Process let $(x = t)$ in $P$ means that every occurrence of $x$ binding $t$ in process $P$.

Process letor $(x = t_1)$ or $(x = t_2)$ in $P$ says that $x$ is bounded to $t_1$ or $t_2$ in process $P$. Note that the or construct does not work in an exclusive way but it can be seen as an union/disjunction. The detailed discussion of the translation rules can be found in the section 9.4 (titled “From protocol specification to logic rules”) of my longer report [Th08, 2012].

Process if nbr$(y_{nid}, y_{nid})$ then $P$ says if node $y_{nid}$ is a neighbor of node $y_{nid}$ then $P$ will run, otherwise, it gets stuck and stays idle. Note that with this process, $sr$-verif considers both uni and bi-directional communication links.

Process accept$([y, \text{List}])$ broadcasts the ID list $[y, \text{List}]$ in case of all the required verification steps made on the reply are successful. This process is used to signal the acceptance of the returned route.

I define the function $[\text{List}, y]$ appends the ID $y$ to the end of the ID list $[\text{List}]$. I remove the syntax of networks and nodes in the simplified syntax of the $sr$-calculus because I focus definitely on presenting the automated verification method, in which the specification of nodes and networks is not required. As mentioned before, the operation of a honest node has to be provided as input, which can be uniformly defined by the following process $P_{\text{spec}}$:

$$P_{\text{spec}} \overset{\text{def}}{=} \text{!INIT} | \text{!INTERM} | \text{!DEST}$$

Every node can start a route discovery towards any other node, in this case process INIT is invoked. Every node can be an intermediate node, in which case its process INTERM is invoked. Finally, every node can be a target node when process DEST runs. Note that the specific structure of each process depends on the specific routing protocol, nevertheless, in general form they can be modeled in the following way:

\[
\begin{align*}
\text{INIT} & \overset{\text{def}}{=} \text{let } (y_{this} = l_{rec}) \text{ in } ((\text{new ID}) \text{Prot}_{\text{init}} \cdot \text{broadinit}(t_1 \in req) \cdot !\text{Rep}_{\text{init}}). \\
\text{Rep}_{\text{init}} & \overset{\text{def}}{=} \text{letor } (y_{next} = y_{honest}) \text{ or } (y_{next} = l_{att}) \text{ in } \text{PRep}_{\text{init}}. \\
\text{PRep}_{\text{init}} & \overset{\text{def}}{=} \text{recvrep}(y_{this}, t_1 | y_{next}, \text{List} \in rep) \cdot \text{if } \text{nbr}(y_{this}, y_{next}) \text{ then } \text{Prot}_{\text{init}} \cdot \text{accept}([y_{next}, \text{List}]).
\end{align*}
\]

43
ROUTING PROTOCOLS

2. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WIRELESS AD-HOC ROUTING PROTOCOLS

On receiving the request, \( P_{\text{Rep}} \) procedures part !\( \text{Rep} \) if \( \text{nbr}(y_{\text{his}}, l_{\text{src}}) \) then \( \text{Prot}_{\text{Interm}_1} \), broadcast(\( t_{[y_{\text{his}}, l_{\text{req}}]} \))!\( \text{Rep}_{\text{Interm}_1} \).

\( \text{P}_{\text{Rep}_{\text{Interm}_1}} \) \( \overset{\text{def}}{=} \) \( \text{rerecv}(t_{[y_{\text{his}}, l_{\text{req}}]} \) if \( \text{nbr}(y_{\text{his}}, l_{\text{src}}) \) then \( \text{Prot}_{\text{Interm}_1} \), broadcast(\( t_{[y_{\text{his}}, l_{\text{req}}]} \))!\( \text{Rep}_{\text{Interm}_1} \).

Each process \( \text{INIT} \), \( \text{INTERM} \) and \( \text{DEST} \) is composed of two parts: (i) A request part, which is placed before the process of form !\( \text{Rep} \), and specifies how requests are handled; and (ii) a reply part !\( \text{Rep} \), which describes how a node behaves when a reply is received. The protocol dependent procedures \( \text{Prot}_{\text{Init}} \), \( \text{Prot}_{\text{Interm}} \), \( \text{Prot}_{\text{Dest}} \) and their \( \text{Prot}_{\text{Rep}} \) counterparts include neighbor checking processes of form if \( \text{nbr}(y_{\text{oid}}, l_{\text{src}}) \) then, functions and “inverse” functions. The notations \( t_{[\text{List}, y_{\text{req}}]} \) and \( t_{[\text{List}, y_{\text{req}}]} \) represent the request and reply messages which include the ID list \([\text{List}]\), respectively.

Process \( \text{INIT} \) considers the case when node \( y_{\text{his}} \) is the source, which is illustrated as \( \text{Scenario1} \) in the Figure 2.17. First, a new message ID is created, then a protocol dependent processing part \( \text{Prot}_{\text{Init}} \) is invoked (e.g., generating a message authentication code in case of the SRP protocol, etc.). If all requirements of the protocol are fulfilled, an initial request \( t_{[\text{List}, y_{\text{req}}]} \) is broadcast, which is specified by process \( \text{broadcast}(t_{[\text{List}, y_{\text{req}}]} \). Afterwards, the source permanently waits for the corresponding reply, which is modeled by process !\( \text{Rep}_{\text{Init}} \). In \( \text{Rep}_{\text{Init}} \) process letor says that the first ID of the ID list included in the reply can be either a honest node or an attacker node. Process \( \text{Prot}_{\text{Init}} \) says that on receiving some reply the source node checks if it is the addressee and the first ID in the list belongs to its neighbor, if so, the reply is processed in a protocol specific manner and the ID list included in the reply is accepted in case the reply fulfills all the requirements.

Process \( \text{INTERM} \) specifies the case when node \( y_{\text{his}} \) is an intermediate node, and is divided to two subprocesses \( \text{Interm}_1 \) and \( \text{Interm}_2 \). These two processes distinguish four different scenarios \( \text{Scenario3-6} \), which are shown in the Figure 2.17.

Process \( \text{Interm}_1 \) describes the case when the current node \( y_{\text{his}} \) receives the initial request. On receiving the request, \( y_{\text{his}} \) examines whether the source is its neighbor continued by rebroadcasting a request when all the protocol specific verification steps are passed, and waits for
the reply. Process $\text{Rep}_{\text{interm}_1}$ is splitted into two processes $\text{Rep}_{\text{interm}_1}$ and $\text{Rep}_{\text{interm}_2}$, where the first one specifies the Scenario 4 in the Figure 2.17 in which both the source and the destination are neighbors of $y_{\text{this}}$, whilst the second subprocess is concerned with the Scenario 5 in which the destination node is not a neighbor of $y_{\text{this}}$ but the source.

Process $\text{Interm}_2$ specifies the scenarios 3 and 6 in the Figure 2.17. In this case the request part is similar as in $\text{Interm}_1$, however, this time there is at least one ID previous $y_{\text{this}}$ in the ID list. Again, $\text{Rep}_{\text{interm}_2}$ is composed of two subprocesses $\text{Rep}_{\text{interm}_2}$ and $\text{Rep}_{\text{interm}_2}$. The first one is concerned with Scenario 3 and the second process represents Scenario 6.

Finally, process $\text{DEST}$ specifies the scenario 2 in the Figure 2.17. Although this scenario involves all the other five scenarios, it is required to be considered because it differs from the others in that it is from the destination’s point of view. When a request is received, the destination checks if the last ID belongs to its neighbor, which is modeled by the part $\text{if } \text{nbr}(y_{\text{this}}, y_{\text{prv}}) \text{ then }$. Thereafter, additional protocol specific processing could be performed. At the end, if the request fulfills all the requirements, then the destination sends back a reply.

One of the advantages of my method is that it requires the user to specify only the operation of a honest node, that is, the operation of the routing protocol which is usually very short. The user is not bothered with specifying the large number of scenarios that involve the attacker nodes. The automated verification is performed on the logic rules that specify the behavior of the protocol. Hence, from the protocol specification in the simplified $sr$-calculus, we have to perform translations into logic rules. The received logic rules should conform the specification of the protocol. All of the possible scenarios will be derived by the translation procedure, which is performed automatically. The overview of the translation procedure can be found in Section 2.8.3.

### 2.8.3 From protocol specification to logic rules

The automatic reasoning procedure is performed on the set of logic rules that represent a source routing protocol, and the attacker’s computation ability and initial knowledge: I refer to the first subset of rules as protocol rules and to the second part as attacker rules. The logic rules used in my method are Horn-clauses known and defined in the logic programming. A Horn-clause basically defines a logic rule $F_1 \land \ldots \land F_n \rightarrow C$, where $F_i$ is a fact while $C$ is a conclusion, and it says that if $F_1, \ldots, F_n$ hold at the same time, then we get $C$. In this section, I show how the logic rules are automatically derived from the protocol specification. First, I introduce the syntax of the logic rules.

### 2.8.4 Syntax of the logic rules

The syntax of the logic rules used in the automatic reasoning is composed of patterns and facts. Patterns correspond to terms in the calculus, and model the elements of request and reply messages. Each term in the calculus has a corresponding pattern in logic. I define a set $E$ of mappings \{\(t_1 \mapsto t^p_1\), \ldots, \(t_m \mapsto t^p_m\)\}, which maps each term to the corresponding pattern.

$\begin{align*}
\text{tp} &:= \text{rreq}^p \mid \text{rrep}^p \mid ID^p \mid n^p [t^p_1, \ldots, t^p_k] \mid \text{tp}^p \mid \text{tp}_{\text{src}} \mid \text{tp}_{\text{dest}} \mid \text{tp}_{\text{att}} \mid x^p \mid \text{yp}^p \mid y_{\text{this}}^p \mid y_{\text{prv}}^p \mid y_{\text{nxt}}^p \mid y_{\text{honprv}}^p \mid y_{\text{honnxt}}^p \mid y_{\text{hid}}^p \mid f(t^p_1, \ldots, t^p_k).
\end{align*}$
2. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WIRELESS AD-HOC ROUTING PROTOCOLS

The patterns $\text{recvreq}$, $\text{recvrep}$, $\text{t_p}_{\text{src}}$, $\text{t_p}_{\text{dest}}$, $\text{att}_n$, and $\text{t}_p_i$, $i \in \{1, \ldots, k\}$, $\pi_{\text{type}}$ represent the same things as the analogous terms in $\pi$-calculus. I adopt the type system of the $\pi$-calculus. $\text{y}_{\text{his}}$, $\text{y}_{\text{prev}}$, $\text{y}_{\text{next}}$ represent variables of type node ID, thus, both the IDs of the attacker and the honest node can be bounded to them. The variables $\text{y}_{\text{his}}^{\text{p}}$, $\text{y}_{\text{prev}}^{\text{p}}$ and $\text{y}_{\text{next}}^{\text{p}}$ defines the variables of type honest node’s ID, hence, only $\text{t}_p_i$ can be bounded. I add the attacker node ID type, $\text{T}_{\text{att}}$, and let $\text{t}_p_i$, be the variable of type $\text{T}_{\text{att}}$, such that only $\text{t}_p_i$, can be bounded to $\text{t}_p_i$.

Names $n$ and function $f$ in the calculus are encoded as functions with arity $k$ and $r$: $n^{\pi}[t_1^p, \ldots, t_k^p]$ and $f(t_1^p, \ldots, t_i^p)$, respectively. In $n^{\pi}[t_1^p, \ldots, t_k^p]$, terms $t_1^p, \ldots, t_k^p$ are the messages that are bounded to parameter in the $\text{recvreq}$, $\text{recvrep}$ actions that occur before the point when new data $n^{\pi}$ is created (i.e., when process (new $n$) in $P$ is called). The reason why I consider $n$ as $n^{\pi}[t_1^p, \ldots, t_k^p]$ is that by using the parameters $t_1^p, \ldots, t_k^p$ we can distinguish different newly created data. $\text{ID}_{\text{rep}}$ represents a session identifier, different session identifiers are associated to each session of processes under replication. $\text{P}[\cdot]$ models a new node ID that has not occurred before. The empty list of parameters, [], attached to $\text{P}$, means that they are new data, which do not depend on the history. The Horn-clauses use the following facts:

$$F := \text{wm}(t^p) \mid \text{att}(\text{t}_p^{\text{att}}, t^p) \mid \text{nbr}(\text{y}_{\text{prev}}^p, \text{y}_{\text{next}}^p) \mid \text{accept}([]) \mid \epsilon.$$  

The meaning of each fact is as follows: $\text{wm}(t^p)$ says that the wireless medium knows $t^p$, which happens when $t^p$ is output by some node; $\text{att}(\text{t}_p^{\text{att}}, t^p)$ says that the attacker $\text{t}_p^{\text{att}}$, knows message $t^p$, which happens when $t^p$ is intercepted or generated by the attacker. Recall that $\text{t}_p^{\text{att}}$ is the variable to which only the constants $\text{t}_p_i^{\text{att}}$, (with different $i$) can be bounded. The fact $\text{nbr}(\text{y}_{\text{prev}}^p, \text{y}_{\text{next}}^p)$ says that node $\text{y}_{\text{next}}^p$ is a neighbor (within the transmission range) of node $\text{y}_{\text{prev}}^p$. Recall that the nbr-fact $\text{nbr}(\text{t}_p^{\text{att}}, \text{t}_p^{\text{att}})$ is not considered during the verification because I do not assume direct links between the attacker nodes.

In the rest of the dissertation, I refer to these facts as $\text{wm}$-facts, $\text{att}$-facts and $\text{nbr}$-facts, respectively. The special fact $\text{accept}([])$ is derived when the reply has got back to the source node that accepts the route specified by the ID list $[]$. The fact $\epsilon$ is used to represent the successful derivation of a given fact during the deduction procedure. If the fact $\epsilon$ is reached, it means that the current derivation branch terminates successfully. The operation of routing protocols is modeled by Horn-clauses, that is, the logic rules $R_1$ of form $F_1 \land \ldots \land F_n \rightarrow C$, where either $F_i$ or $C$ are one of the fact given above, and $\land$ represents a logical AND (i.e., conjuction). Note that $n$ can be zero, which means that $R$ has a form of $C$. The left side of $R$ is called as hypothesis whilst the right side is called as conclusion.

Due to page limitation, here I only provide a short overview, the full list of the translation rules can be found in the section 9.4 (titled “From protocol specification to logic rules”) of my longer report [Th08, 2012].

2.8.5 The resulting protocol rules

Based on the translation procedure (detailed in the section 9.4 of my report [Th08, 2012]), the translation of the protocol specification $P_{\text{spec}}$ given in Section 2.8.2, yields the following logic rules, after sanitizing (i.e., eliminating duplications and redundancy).

(Protocol rules: Template of the correct operation) :=

\[
\begin{align*}
R_1^\text{p} := & \text{wm}(t^p_{\text{req}}). \\
R_2^\text{p} := & \text{wm}(t^p_{\text{rep}}) \land \text{nbr}(t^p_{\text{src}}, y^p_{\text{his}}) \land \text{nbr}(y^p_{\text{his}}, t^p_{\text{src}}) \land \text{Prot}_i^{\text{facts}} \rightarrow \text{wm}(t^p_{\text{req}}). \\
R_3^\text{p} := & \text{wm}(t^p_{\text{req}}) \land \text{nbr}(t^p_{\text{src}}, t^p_{\text{att}}) \rightarrow \text{att}(t^p_{\text{att}}, t^p_{\text{req}}). \\
R_4^\text{p} := & \text{wm}(t^p_{\text{req}}) \land \text{nbr}(y^p_{\text{prev}}, y^p_{\text{his}}) \land \text{nbr}(y^p_{\text{his}}, y^p_{\text{next}}) \land \text{Prot}_i^{\text{facts}} \rightarrow \text{wm}(t^p_{\text{req}}). \\
R_5^\text{p} := & \text{wm}(t^p_{\text{req}}) \land \text{nbr}(y^p_{\text{prev}}, y^p_{\text{his}}) \land \text{nbr}(y^p_{\text{his}}, t^p_{\text{att}}) \rightarrow \text{att}(t^p_{\text{att}}, t^p_{\text{req}}). \\
R_6^\text{p} := & \text{att}(t^p_{\text{att}}, t^p_{\text{req}}) \land \text{nbr}(t^p_{\text{att}}, t^p_{\text{src}}) \land \text{nbr}(y^p_{\text{his}}, t^p_{\text{att}}) \rightarrow \text{att}(t^p_{\text{att}}, t^p_{\text{req}}). \\
R_7^\text{p} := & \text{att}(t^p_{\text{att}}, t^p_{\text{req}}) \land \text{nbr}(t^p_{\text{att}}, t^p_{\text{src}}) \land \text{nbr}(y^p_{\text{his}}, t^p_{\text{att}}) \land \text{Prot}_i^{\text{facts}} \rightarrow \text{wm}(t^p_{\text{req}}). \\
R_8^\text{p} := & \text{att}(t^p_{\text{att}}, t^p_{\text{req}}) \land \text{nbr}(t^p_{\text{att}}, t^p_{\text{src}}) \land \text{nbr}(y^p_{\text{his}}, t^p_{\text{att}}) \land \text{Prot}_i^{\text{facts}} \rightarrow \text{wm}(t^p_{\text{req}}). \\
\end{align*}
\]
\[R^p_{1.1}\] \(\text{wm}(t_{\text{req}}^p) \land \text{nbr}(y_{\text{protv}}, l_{\text{dest}}^p) \land \text{nbr}(l_{\text{dest}}^p, y_{\text{protv}}) \land \text{Profacts} \rightarrow \text{wm}(y_{\text{protv}}, t_{\text{rep}}^p)\).

\[R^p_{1.2}\] \(\text{wm}(t_{\text{req}}^p) \land \text{nbr}(y_{\text{protv}}, l_{\text{dest}}^p) \land \text{nbr}(l_{\text{dest}}^p, y_{\text{protv}}) \land \text{nbr}(l_{\text{dest}}^p, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{att}(l_{\text{att}}^p, y_{\text{protv}}, t_{\text{rep}}^p)\).

\[R^p_{1.3}\] \(\text{att}(l_{\text{att}}^p, t_{\text{req}}^p) \land \text{nbr}(l_{\text{att}}^p, l_{\text{dest}}^p) \land \text{nbr}(l_{\text{dest}}^p, y_{\text{protv}}) \land \text{nbr}(l_{\text{dest}}^p, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{att}(l_{\text{att}}^p, y_{\text{protv}}, t_{\text{rep}}^p)\).

\[R^p_{1.4}\] \(\text{att}(l_{\text{att}}^p, t_{\text{req}}^p) \land \text{nbr}(l_{\text{att}}^p, l_{\text{dest}}^p) \land \text{Profacts} \rightarrow \text{att}(l_{\text{att}}^p, l_{\text{dest}}^p, t_{\text{rep}}^p)\).

\[R^p_{1.5}\] \(\text{att}(l_{\text{att}}^p, t_{\text{req}}^p) \land \text{nbr}(l_{\text{att}}^p, l_{\text{dest}}^p) \land \text{Profacts} \rightarrow \text{att}(l_{\text{att}}^p, y_{\text{protv}}, t_{\text{rep}}^p)\).

\[R^p_{2.1}\] \(\text{RequestPart} \land \text{wm}(y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{src}}^p) \land \text{Profacts} \rightarrow \text{wm}(l_{\text{src}}^p, t_{\text{rep}}^p)\).

\[R^p_{2.2}\] \(\text{RequestPart} \land \text{wm}(y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{src}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{att}(l_{\text{att}}^p, (l_{\text{src}}^p, t_{\text{rep}}^p))\).

\[R^p_{2.3}\] \(\text{RequestPart} \land \text{att}(l_{\text{att}}^p, y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{nbr}(y_{\text{this}}, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{accept}(y_{\text{protv}}, l_{\text{att}}^p, Listf)\).

\[R^p_{2.4}\] \(\text{RequestPart} \land \text{wm}(y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{Profacts} \rightarrow \text{wm}(l_{\text{src}}^p, t_{\text{rep}}^p)\).

\[R^p_{2.5}\] \(\text{RequestPart} \land \text{att}(l_{\text{att}}^p, y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{nbr}(y_{\text{this}}, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{wm}(l_{\text{src}}^p, t_{\text{rep}}^p)\).

\[R^p_{2.6}\] \(\text{RequestPart} \land \text{wm}(y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{nbr}(y_{\text{this}}, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{wm}(l_{\text{src}}^p, t_{\text{rep}}^p)\).

\[R^p_{2.7}\] \(\text{RequestPart} \land \text{att}(l_{\text{att}}^p, y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{nbr}(y_{\text{this}}, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{wm}(l_{\text{src}}^p, t_{\text{rep}}^p)\).

\[R^p_{2.8}\] \(\text{RequestPart} \land \text{wm}(y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{Profacts} \rightarrow \text{wm}(l_{\text{src}}^p, t_{\text{rep}}^p)\).

\[R^p_{2.9}\] \(\text{RequestPart} \land \text{att}(l_{\text{att}}^p, y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{nbr}(y_{\text{this}}, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{wm}(l_{\text{src}}^p, t_{\text{rep}}^p)\).

\[R^p_{2.10}\] \(\text{RequestPart} \land \text{att}(l_{\text{att}}^p, y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{nbr}(y_{\text{this}}, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{wm}(y_{\text{protv}}, t_{\text{rep}}^p)\).

\[R^p_{2.11}\] \(\text{RequestPart} \land \text{att}(l_{\text{att}}^p, y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{nbr}(y_{\text{this}}, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{wm}(y_{\text{protv}}, t_{\text{rep}}^p)\).

\[R^p_{2.12}\] \(\text{RequestPart} \land \text{att}(l_{\text{att}}^p, y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{nbr}(y_{\text{this}}, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{wm}(y_{\text{protv}}, t_{\text{rep}}^p)\).

\[R^p_{2.13}\] \(\text{RequestPart} \land \text{att}(l_{\text{att}}^p, y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{nbr}(y_{\text{this}}, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{wm}(y_{\text{protv}}, t_{\text{rep}}^p)\).

\[R^p_{2.14}\] \(\text{RequestPart} \land \text{att}(l_{\text{att}}^p, y_{\text{this}}, t_{\text{rep}}^p) \land \text{nbr}(y_{\text{this}}, l_{\text{dest}}^p) \land \text{nbr}(y_{\text{this}}, y_{\text{protv}}) \land \text{nbr}(y_{\text{this}}, l_{\text{att}}^p) \land \text{Profacts} \rightarrow \text{wm}(y_{\text{protv}}, t_{\text{rep}}^p)\).
2. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WIRELESS AD-HOC ROUTING PROTOCOLS

\[ R_{2.15}^{req} \land \text{RequestPart} \land \text{wm}((y^p_{\text{this}}, t^p_{\text{rep}})) \land \text{nbr}(y^p_{\text{this}}, y^p_{\text{att}}) \land \text{nbr}(y^p_{\text{this}}, l^p_{\text{src}}) \rightarrow \text{wm}((y^p_{\text{honprv}}, t^p_{\text{rep}})) \]

\[ R_{2.16}^{rep} \land \text{RequestPart} \land \text{att}(t^p_{\text{att}}, (y^p_{\text{this}}, t^p_{\text{rep}})) \land \text{nbr}(l^p_{\text{src}}, y^p_{\text{this}}) \land \text{nbr}(y^p_{\text{this}}, l^p_{\text{att}}) \land \text{nbr}(y^p_{\text{this}, \text{acts}}) \rightarrow \text{wm}((y^p_{\text{honprv}}, t^p_{\text{rep}})) \]

Again, I note that these rules consider a general case, the protocol dependent processing parts, and the request and reply messages \( t^p_{\text{req}} \) and \( t^p_{\text{rep}} \) are defined by a specific protocol. Figure 2.18, Figure 2.19 and Figure 2.20 illustrate the scenarios described by each logic rule. In the figures, the nodes labeled by \( l_{\text{src}}, l_{\text{dest}} \) and \( l_{\text{att}} \) are the source, the destination and the attacker nodes. The remaining nodes represent honest intermediate nodes. The arrow with the label \( \text{req} \) or \( \text{rep} \) represents the outputing of request or reply messages. The direction of arrows is from the sender node to the receiver node. For example, the edge with label \( \text{req} \) from node A to node B means that B has received the request or reply sent by A. The arrow with label \( \text{nbr} \) represents the neighbor check performed by a node after receiving some request or reply. For instance, the edge labeled by \( \text{nbr} \) from \( l^p_{\text{src}} \) to \( l^p_{\text{att}} \) means that the source node checks whether the attacker is its neighbor. The arrow with label \( (\text{nbr}, \text{req}) \) or \( (\text{nbr}, \text{rep}) \) has the joint interpretation of the previous two cases, that is, the neighbor check was carried out and the reply or request is forwarded.

To make this more clear I provide the interpretation of some scenarios in the Figures: In \( R_{2.1}^{req} \), after receiving a request broadcast by the source, \( y_{\text{this}} \) checks if \( l_{\text{src}} \) is its neighbor, if so, it re-broadcasts the request. \( R_{2.1}^{req} \) present the case when the attacker impersonates \( l_{\text{src}} \), and says that after \( y_{\text{this}} \) receives the request sent by \( l_{\text{att}} \) it checks if the source is its neighbor. Finally, \( R_{2.2}^{req} \) says that after performing neighbor checks, \( y_{\text{this}} \) forwards the reply addressed to the source node, however, this reply is overheared by the attacker. Other rules can be interpreted in a similar way.

![Figure 2.18: The scenarios corresponding to the logic rules which are concerned with the request phase.](image-url)
the initial request. Rules $R_{2.3}^{req}$ and $R_{2.4}^{req}$ are similar to the previous two rules, but this time the sender is some honest intermediate node $y_{honprv}$. Rules $R_{2.5}^{req}$ and $R_{2.6}^{req}$ are related to the scenarios when the attacker node forwards some request: The first rule considers the case when the attacker follows the protocol faithfully and appends its ID to the ID list, whilst the second is concerned with the case when the attacker impersonates $y_{honprv}$. $R_{2.7}^{req}$ is similar to $R_{2.6}^{req}$ except that in $R_{2.7}^{req}$, the attacker impersonates the source node.

The five rules $R_{3.1}^{req}$, . . . , $R_{3.5}^{req}$ define the scenarios from the destination’s point of view, and describe how the destination processes request messages. In $R_{3.1}^{req}$, node $y_{honprv}$ gets back the reply addressed to it from the destination. In $R_{3.2}^{req}$, the reply addressed to $y_{this}$ is overheared by the attacker. Rule $R_{3.3}^{req}$ is associated with the case when the attacker impersonates $y_{honprv}$. In $R_{3.4}^{req}$, the attacker follows the protocol when it sends a request to $l_{dest}$. $R_{3.5}^{req}$ is the same to $R_{3.3}^{req}$ but this time the attacker is not within the transmission range of the destination.

The rules $R_{2.1}^{rep}$, . . . , $R_{2.16}^{rep}$ describe how an intermediate node $y_{this}$ handles reply messages. The first three rules specify the scenarios in which $y_{this}$ receives a reply from the destination. In all the three cases $R_{2.1}^{rep}$, $R_{2.2}^{rep}$, and $R_{2.3}^{rep}$, $y_{this}$ checks if $l_{dest}$ and the ID ($y_{honprv}$ or $l_{att}$) before $y_{this}$ in the list included in the reply belongs to its neighbors, in that case $y_{this}$ forwards the reply to node $y_{honprv}$ or $l_{att}$, respectively. The next three rules $R_{2.4}^{rep}$ - $R_{2.6}^{rep}$ are the same to the previous three rules except that $l_{dest}$ is replaced by some honest intermediate node $y_{honext}$. $R_{2.7}^{rep}$ and $R_{2.16}^{rep}$ describe the case when the ID after $y_{this}$ is $l_{att}$. Rules $R_{2.8}^{rep}$ - $R_{2.10}^{rep}$ are the same to the rules $R_{2.4}^{rep}$ - $R_{2.6}^{rep}$ but they involve an intermediate node instead of $l_{src}$. Rules $R_{2.12}^{rep}$ - $R_{2.14}^{rep}$ involve only intermediate nodes. Eventually, in $R_{2.11}^{rep}$ and $R_{2.15}^{rep}$ the attacker node is the addressee of the reply.

RequestPart in each reply rule considers the corresponding request of the reply, such that whenever a reply is received by some node, it is accepted only in case its corresponding request has been broadcast before. The part ProtLacts in the rules corresponds to the processes Prot and ProtRep in the calculus specification. They represent additional protocol specific processing steps, which usually are additional neighbor checks or execution of some function such as signature check, decryption, computing of hash and MAC values. One advantage of my method is that it is based resolution which includes the application of unification methods. Unification can be used to model implicitly the verification procedure such as MAC, hash and signature verifications. Therefore only additional neighbor checks should be explicitly defined, which means that ProtLacts can either be composed of nbr-facts or can be empty. More details about unification and resolution can be found in Section 2.8.7.
2.8.6 Specifying the attacker rules

The ability of a compromised node is represented in the following rules.

\[ (\text{Init. knowl.}) := \forall p \in \text{neighbors of } l_{\text{att}}^p \]
\[ I_{\text{att}ID}^{\text{own}} \cdot \text{att}(l_{\text{att}}^p, l_{\text{att}}^p) \rightarrow \text{att}(l_{\text{att}}^p, l_{\text{att}}^p); I_{\text{att}Key}^p \cdot \text{att}(k(l_{\text{att}}^p, l_{\text{att}}^p)) \]
\[ I_{\text{att}ID}^{\text{pKey}} \cdot \text{att}(pk(l_{\text{att}}^p)) \text{ for all honest } l_{\text{att}}^p \]

\[ I_{\text{att}ID}^{\text{pKey}} \text{ and } I_{\text{attID}}^{\text{pKey}} \text{ mean that initially the attacker knows its own ID and the IDs of its honest neighbors, respectively. } I_{\text{att}Key} \text{ and } I_{\text{att}ID}^{\text{pKey}} \text{ say that the attacker possesses all the keys it shares with the honest nodes, and all public keys. Function } pk(l_{\text{att}}^p) \text{ represents the public key of node } l_{\text{att}}^p. \]
\[ I_{\text{att}Key} \quad \text{and} \quad I_{\text{attID}}^{\text{pKey}} \quad \text{in addition, I define a computation ability for the attacker node as follows:} \]

\[ (\text{Computation ability - protocol independent}) := \]
\[ A_{\text{data}} \cdot \text{att}(l_{\text{att}}^p, s) \rightarrow \text{att}(l_{\text{att}}^p, n^p[s]) \]
\[ A_{\text{auth}} \cdot \text{For each public function } f \text{ of n-arity} \]
\[ \text{att}(l_{\text{att}}^p, x_1^p) \land \cdots \land \text{att}(l_{\text{att}}^p, x_n^p) \rightarrow \text{att}(l_{\text{att}}^p, f(x_1^p, \ldots, x_n^p)) \]
\[ A_{\text{hash}} \cdot \text{att}(l_{\text{att}}^p, x^p) \rightarrow \text{att}(l_{\text{att}}^p, h(x^p)) \]
\[ A_{\text{dense}} \cdot \text{att}(l_{\text{att}}^p, x^p) \land \text{att}(l_{\text{att}}^p, k^p(y^p_{\text{nid}}, y^p_{\text{nid}})) \rightarrow \text{att}(l_{\text{att}}^p, \text{enc}(x^p, k^p(y^p_{\text{nid}}, y^p_{\text{nid}}))) \]
\[ A_{\text{penc}} \cdot \text{att}(l_{\text{att}}^p, x^p) \land \text{att}(l_{\text{att}}^p, \text{pk}(y^p_{\text{nid}})) \rightarrow \text{att}(l_{\text{att}}^p, \text{penc}(x^p, \text{pk}(y^p_{\text{nid}}))) \]
\[ A_{\text{sign}} \cdot \text{att}(l_{\text{att}}^p, x^p) \land \text{att}(l_{\text{att}}^p, \text{sk}(y^p_{\text{nid}})) \rightarrow \text{att}(l_{\text{att}}^p, \text{sign}(x^p, \text{sk}(y^p_{\text{nid}}))) \]
\[ A_{\text{mac}} \cdot \text{att}(l_{\text{att}}^p, x^p) \land \text{att}(l_{\text{att}}^p, \text{sk}(y^p_{\text{nid}}, y^p_{\text{nid}})) \rightarrow \text{att}(l_{\text{att}}^p, \text{mac}(x^p, \text{sk}(y^p_{\text{nid}}, y^p_{\text{nid}}))) \]
\[ A_{\text{add}} \cdot \text{att}(l_{\text{att}^p}, \text{List}^p) \land \text{att}(l_{\text{att}^p}, y^p) \rightarrow \text{att}(l_{\text{att}^p}, \text{List}^p, y^p) \]
\[ A_{\text{comp}} \cdot \text{att}(l_{\text{att}}, l_{\text{att}}, t^p_1) \land \cdots \land \text{att}(l_{\text{att}}, l_{\text{att}}, t^p_m) \rightarrow \text{att}(l_{\text{att}}, l_{\text{att}}, t^p_1, \ldots, t^p_m) \]
\[ A_{\text{dec}} \cdot \text{att}(l_{\text{att}^p}, t^p_1, \ldots, t^p_m) \text{ derives additional m rules:} \]
\[ \text{att}(l_{\text{att}}, l_{\text{att}}, t^p_1, \ldots, t^p_m) \rightarrow \text{att}(l_{\text{att}}, l_{\text{att}}, t^p_1, \ldots, t^p_m) \]
\[ A_{\text{msg}} \cdot \text{att}(l_{\text{att}}, (\text{head}; v_1; \ldots; \text{List}; \ldots; v_k)) \text{ yields additional } k+2 \text{ rules} \]
\[ \text{att}(l_{\text{att}}, (\text{head}; v_1; \ldots; \text{List}; \ldots; v_k)) \rightarrow \text{att}(l_{\text{att}}, (\text{head});) \]
\[ \vdots \]
\[ \text{att}(l_{\text{att}}, (\text{List};) \ldots; v_k)) \rightarrow \text{att}(l_{\text{att}}, (\text{List});) \]
\[ \vdots \]
\[ \text{att}(l_{\text{att}}, (\text{List};) \ldots; v_k)) \rightarrow \text{att}(l_{\text{att}}, (\text{List});) \]
\[ \vdots \]
composes them to smaller parts. I define reply and request as the form (patterns it deliberately considers request and reply messages, which have a specific form, and de-

\[\begin{align*}
A_{\text{List}}^{\text{contain}}: & \quad \text{att}(t^p_{\text{att}}, [v^p_i], \text{List}) \rightarrow \text{att}(t^p_{\text{att}}, \text{List})

A_{\text{replace}}: \quad & \text{att}(t^p_{\text{att}}, [v^p_i], \text{List}) \rightarrow \text{att}(t^p_{\text{att}}, \text{List})

A_{\text{replace}}^{\text{ID}_1}: \quad & \text{att}(t^p_{\text{att}}, \text{rrep/rreq}, v^p_{\text{src}}, \text{ID}, t^p_{\text{att}}, [\text{List}]) \rightarrow \text{att}(t^p_{\text{att}}, \text{ID}, [\text{List}])

A_{\text{replace}}^{\text{ID}_2}: \quad & \text{att}(t^p_{\text{att}}, \text{rrep/rreq}, v^p_{\text{src}}, \text{ID}, t^p_{\text{att}}, [\text{List}]) \rightarrow \text{att}(t^p_{\text{att}}, \text{ID}, [\text{List}])

\vdots

A_{v_k}^{\text{replace}}: \quad & \text{att}(t^p_{\text{att}}, \text{rrep/rreq}, v^p_{\text{src}}, \text{ID}, t^p_{\text{att}}, [\text{List}]) \rightarrow \text{att}(t^p_{\text{att}}, [\text{List}])
\end{align*}\]

Let the set of all attacker’s rules be \( C_{\text{att}} \). Rule \( A_{\text{data}} \) says that the attacker node can create arbitrary new data \( v^p \) such as fake ID identifiers, where \( s \) is a session ID to identify the session in which data was generated. Rule \( A_{\text{fun}} \) says that if the attacker is aware of \( x^p_1, \ldots, x^p_n \), it can compute some function \( f \) on them. Depending on the value of \( f \) the following rules are defined: \( A_{\text{hash}} \) says that if the attacker has some pattern \( x^p \) it can compute its hash value \( h(x^p) \). Rule \( A_{\text{sign}} \) and \( A_{\text{mac}} \) say that the attacker can encrypt some message \( x^p \) it has with a shared key or a public key it possesses, respectively. Rules \( A_{\text{sign}} \) and \( A_{\text{mac}} \) say that the attacker is able to sign some message with the private key it has, and it can compute a message authentication code of some message using the shared key it possesses, respectively. \( A_{\text{odd}} \) says that the attacker can append a node ID to the end of the list \( [\text{List}] \), where \( \text{List} \) can be empty. Finally, rule \( A_{\text{comp}} \) is a composition rule, which says that the attacker is capable to compose a tuple of the patterns it owns.

Rule \( A_{\text{dom}} \) is the decomposition rule, which says that if the attacker has a tuple of \( m \) elements, it has each element as well. Note that due to hash function is one-way, it has no corresponding inverse function. Rule \( A_{\text{msg}} \) is similar to \( A_{\text{dom}} \) but instead of decomposing a tuple of arbitrary patterns it deliberately considers request and reply messages, which have a specific form, and decomposes them to smaller parts. I define reply and request as the form \((\text{head}, v_1; \ldots; \text{List}; \ldots; v_k)\), which is usually valid in case of source routing protocols. The part \( \text{head} \) represents the header \((\text{rrep/rreq}, v^p_{\text{src}}, v^p_{\text{dest}}, \text{ID})\) of reply and request messages, respectively. The \( \text{List} \) is the ID list included in a reply or a request, and the \( k \) elements \( v_1, \ldots, v_k \) are the remaining parts. Each \( v_i \) can be an “standalone” encryption, signature, hash, and MAC, or it can be the tuple of them. For instance, in the SRP protocol \( v_i \) is a MAC, while in the Ariadne protocol \( v_i \) can be a tuple of several signatures. Finally, rules \( A_{\text{contain}}^{\text{List}_1} \) and \( A_{\text{contain}}^{\text{List}_2} \) say that if the attacker has an ID list \([\text{List}, v^p_i] \) and \([v^p_i], \text{List} \), respectively, where \( v^p_i \) is new node ID that has not occurred before, during the verification procedure, then the attacker has the sublist \([\text{List}] \). Let the set of all \( A_{\text{contain}}^{\text{List}} \)-type rules be \( \text{CONTAIN} \). The rationale behind the \( \text{CONTAIN} \) rules is to reason about the source routing protocols where the ID list included in each request/reply message determines the structure of the message (e.g., the per-hop hash and signatures defined in Ariadne, endAirA). We can add more rules into \( \text{CONTAIN} \) to examine the case when the new ID is inserted to every possible places in the ID list. Rules \( A_{\text{replace}}^{\text{ID}_1}, A_{\text{ID}_2} \) say that if the attacker has a request/reply in which an incorrect typed (i.e., not a node ID type) data, \( v^p_{\text{incorrect}} \), is located in the place of the node ID, then the attacker has \( v^p_{\text{incorrect}} \). Let the set of all \( A_{\text{replace}}^{\text{ID}} \)-type rules be \( \text{REPLACE} \). The difference between the rules in \( A_{\text{msg}} \) and \( A_{\text{replace}} \) is that in \( A_{\text{msg}} \) the message parts are located in correct places within the request/reply. Namely, the type of the expected data at a given place within a request or a reply matches the type of the data which is residing in that place.

2.8.7 Automating the verification using resolution-based deduction and backward searching

In this section, I present an automatic verification technique based on resolution-based deduction. Before discussing the algorithm some notions and definitions are introduced.

2.8.8 Derivation

**Resolution:** The verification is based on a guided execution of resolution steps. The notion of resolution [61] is also known in logic programming and is applied in broadly used languages such as Prolog. Intuitively, a resolution step can be seen as sequential execution of two logic rules,
and yields a new rule or a fact. To understand the formal definition of resolution, first, I review
the notion of substitution and unification (known in logic programming). Substitution binds some
pattern to some variable, and it is typically denoted by $\sigma$. For instance, $\sigma = \{ y_{mid} \leftarrow v^*_{src} \}$ binds
constant $v^*_{src}$ to the variable $y_{mid}$. I note that $\sigma$ can be more complex and contains a lot of bindings.

Unification is defined over a set of facts, and intuitively brings facts into a common form. More
precisely, $F_1$ and $F_2$ are unifiable if there exist some $\sigma$ such that $F_1\sigma = F_2\sigma$, where $F\sigma$ represents
the application of $\sigma$ to $F$. The substitution $\sigma$ that unifies two facts is called as an unifier.

As mentioned at the end of Section 2.8.5, verification procedures such as hash and MAC verifica-
tion are implicitly modeled by unification. To illustrate this let us take the following example rule:

\[
wm((l^p_{src}, l^p_{1}), MAC(l^p_{src}, k(l^p_{1}, l^p_{dest}))) \rightarrow accept(l^p_{1}))
\]

This rule says that if the source receives a MAC value of list $[l^p_1]$ encrypted with the shared key $k(l^p_{src}, l^p_{dest})$ then it accepts the list. Furthermore, let us say that during the automatic reasoning the fact $wm((l^p_{src}, l^p_{1}), MAC(l^p_{src}, k(l^p_{1}, l^p_{2})))$ has
been derived, which means that a message including the MAC computed with the key $k(l^p_{1}, l^p_{2})$ is
sent to $l^p_{src}$. Of course, because the MAC is not correct the source will not accept the received
message, which formally means that the two $wm$-facts are not unifiable.

**Definition 9.** A substitution $\sigma$ is a most general unifier (mgu) of a set of facts $S$ if it unifies $S$, and for any unifier $\omega$ of $S$, there is a unifier $\lambda$ such that $\omega = \sigma \circ \lambda$.

Two facts may have several unifiers but only one mgu. Now, we have arrived at the point to
provide a formal definition of resolution:

**Definition 10.** Given two rules $r_1 = H_1 \rightarrow C_1$, and $r_2 = F \land H_2 \rightarrow C_2$, where $F$ is any hypothesis of $r_2$, and $F$ is unifiable with $C_1$ with the most general unifier $\sigma$, then the resolution $r_1 \circ_F r_2$ of them yields a new rule $H_1 \land H_2 \sigma \rightarrow C_2\sigma$.

For instance, let $r_1$ says that “if the attacker has a message $m$, then it has the cipher text $enc$”;
and $r_2$ says that “if the attacker has a chipertext $enc$ and a secret key $k$, it gets the plaintext $p$”. In this case, the resolution $r_1 \circ_F r_2$ says that “if the attacker has a message $m$ and a key $k$, it gets the plaintext $p$”. Here, $F$ is the common part that says “if the attacker has a chipertext $enc$”, which is eliminated in the result. Note that both $r_1$ and $r_2$ can be a fact.

**Resolution in a backward manner:** A backward effect is achieved by performing resolution
$R_0 F$ between a rule $R$ and a fact $F$, where $R = F_1 \land \ldots \land F_n \rightarrow C$, and $F$ is unifiable with $C$. As the result of $R_0 F F$, after $F$ and $C$ are unified with $\sigma$, they are eliminated and the hypothesis
of $R$, $F_1 \sigma \land \ldots \land F_n \sigma$, is yielded.

A rule $R$ of form $F_1 \land \ldots \land F_n \rightarrow C$ is illustrated visually as a tree such that the conclusion
$C$ is a root and the leaves are composed of the facts $F_i$, $i \in \{1, \ldots, n\}$. The root is labeled with
the name of the rule. Finally, the edges in the tree are directed edges drawn from the child to its
father. The resolution $R \circ_F F$ is illustrated as a tree, where the unified form of $F$ and $C$, $C\sigma$, resides in the root and the leaves consist of the facts $F_1\sigma, \ldots, F_n\sigma$, where $\sigma$ is the most general unifier of $F$ and $C$. As an example, in Figure 2.21 the trees in case of the rules $A^{hash}, A^{enc}, A^{sign}$, and $R^*_{1,1}$ are given.

The intuition behind the direction of edges lies in the fact that we perform backward searching
during the verification. For instance, if during the verification the next fact to be proved is
$att(l^p_{att}, h(x^p))$: Let us consider the tree corresponds to the rule $A^{hash}$ in Figure 2.21. Intuitively, the
direction of the edge means that in order to be able to compute a one-way hash on $x^p$, the
attacker has to possess $x^p$. The following step would be proving the fact $att(l^p_{att}, x^p)$ and then
continued with proving each of the facts that are required to derive $att(l^p_{att}, x^p)$, and so on.

**Definition 11.** The derivation tree of a fact $F$ is the tree in which the root is $F$ and each level below $F$ specifies such rules which have been applied by the verification algorithm to derive $F$. Let $\epsilon$ be a fact that takes its value from $\{ l_{att}^{p} \cup l_{att}^{q} \cup l_{att}^{r} \cup l_{att}^{eq} \cup R_{1,1}^{eq} \}$. We say that a fact $F$ is derivable using the set of rules $\mathcal{R}$ if there exists a derivation tree of $F$ in which the nodes are labeled with the rules in $\mathcal{R}$, and all the leaves are $\epsilon$.

Intuitively, the derivation tree represents a consecutive execution of resolution steps. Next,
I introduce some notions that I will use when describing the verification algorithm. Whenever
2.8. sr-verif: On automating the verification

![Derivation Trees]

Figure 2.21: The trees correspond to the attacker rules $A^\text{hash}$, $A^\text{mac}$, $A^\text{sign}$, and the protocol rule $R_{1,1}^{\text{rep}}$. The derivation trees corresponding to the attacker rules are filled.

a resolution step $R \circ F$ is executed during the backward search, I say that a derivation tree is extended at the node $F$ with rule $R$, or in other words, the children of node $F$ are computed with $R$. In case $F$ is extended for the first time, I say that $F$ is depth-extended, otherwise, when at some point of the searching procedure the algorithm returns to node $F$ and extend the tree with some another rule, I say that $F$ is breadth-extended.

2.8.9 The verification algorithm: Combining BDSR with logic based resolution

To automate the BDSR algorithm, I apply a resolution based deduction approach. In the following, I give an overview of the algorithm in more details. For simplicity, I start with discussing the backward deduction algorithm for one attacker node. Extending the algorithm in order to handle more than one attacker node is straightforward, which I will discuss at the end of this section. Specifically, the algorithm discussed below refers to the variable $l^p_{\text{att}}$, which represents any attacker node.

Algorithm: The input of the algorithm is the maximal length of the invalid route, $n$. During the verification the algorithm examines the invalid route from length 1 to $n$. The attack scenario is stored in the tuple of three sets ($T_{\text{top}}$, $M_{\text{msg}}$, $A$), where $T_{\text{top}}$ is used to store the “attack topology” in which the attack has been detected, $M_{\text{msg}}$ is the set of the messages exchanged by honest nodes during the attack, and $A$ stores messages sent by the attacker represented by the variable $l^p_{\text{att}}$. At the end, in case an attack scenario is detected the attack scenario ($T_{\text{top}}$, $M_{\text{msg}}$, $A$) is returned, otherwise, ($\emptyset$, $\emptyset$, $\emptyset$) is returned.

At first, $T_{\text{top}}$ includes only two nodes: The destination node $l^p_{\text{dest}}$, and the source node $l^p_{\text{src}}$ without any edge. Whenever new possible edges and nodes (either a honest or an attacker) are found through which the reply or request could have traversed, $T_{\text{top}}$ is updated with them. In addition, the exchanged request and reply messages that are concerned with the discovered route are added to the set $M_{\text{msg}}$, and the attacker behaviour is continually tracked by updating $A$. $T_{\text{top}}$ is updated after each resolution step $R \circ F$, where the hypothesis of $R$ involves at least one $nbr$-fact. In particular, the update of $T_{\text{top}}$ is composed of adding $nbr$-facts in the hypothesis $Hyp$ that is resulted from $R \circ F$ into $T_{\text{top}}$, which is followed by removing the $nbr$-facts from $Hyp$. The update of $M_{\text{msg}}$ and $A$ are performed by adding the remaining $wm$-facts and $att$-facts in $Hyp$, respectively. Note that $wm$-facts and $att$-facts are not removed from $Hyp$ but only the $nbr$-facts. At the end of the updating procedure the three sets are sanitized by eliminating fact duplications in them. The analogous interpretation for derivation tree is as follows: Node $F$ is depth-extended with rule $R$, and $T_{\text{top}}$, $M_{\text{msg}}$, $A$ are updated and sanitized. Finally, the $nbr$-facts among the children of $F$ and their corresponding edges are removed from the tree.
2. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WIRELESS AD-HOC ROUTING PROTOCOLS

The logic rules are classified into four subsets based on the different scenarios they specify. Set $S_{\text{hon}}$ includes the request and reply rules concerning the scenarios in which the messages are exchanged between only honest nodes. $S_{\text{hon}}$ is divided into the disjoint subsets $S_{\text{req}}^{\text{hon}}$ and $S_{\text{rep}}^{\text{hon}}$ for the request and reply phases, respectively. The rules describing the case when the attacker node receives or overhears a message are put into $S_{\text{Recv}}^{\text{att}}$. Set $S_{\text{Fw}}^{\text{att}}$ contains the rules that consider the scenario in which the attacker forwards some message to its neighbors. The rules defining the attacker’s computation ability and initial knowledge are stored in $S_{\text{Comp}}^{\text{att}}$.

Figure 2.22: The figure shows the general derivation tree occur during verification. The rulesets placed at each phase and edges represent that in that phase they are used by the algorithm to extend the tree. The notation $\epsilon \mid X$ says that the derivation can either terminate successfully or gets stuck. The dots after an arrow between each node pair represent a set of edges, which can be empty.

As mentioned before, at the beginning I assume that the fact $\text{accept}([\text{List}_{\text{invalid}}])$ has been derived. The route $[\text{List}_{\text{invalid}}]$ is formally defined by the set $T_{\text{invalid}}$ that is composed of nbr-facts. For example, $[P_l]$ is defined by set \{nbr($l_{\text{src}}$, $P_l$), nbr($P_l$, $l_{\text{src}}$), nbr($l_{\text{dest}}$, $P_l$), nbr($P_l$, $l_{\text{dest}}$)\}. During the verification procedure, whenever the topology $T_{\text{top}}$ is updated, the algorithm checks if $T_{\text{invalid}} \subseteq T_{\text{top}}$ holds. If the answer is yes, we get a contradiction because the route $[\text{List}_{\text{invalid}}]$ is valid in $T_{\text{top}}$, which means that $T_{\text{top}}$ is not an attack topology. Hence, in the description of the algorithm below, by derivation of some fact I mean such a derivation during which $T_{\text{invalid}} \subseteq T_{\text{top}}$ never holds whenever $T_{\text{top}}$ is updated. The rest part of the verification is concerned with searching for a derivation of the fact $\text{accept}([\text{List}_{\text{invalid}}])$. Figure 2.22 shows the general forms of the derivation tree that may occur during the backward searching procedure.

In the honest phases the tree is continually extended using the rules in $S_{\text{hon}}$. Within the attacker phases the tree is extended with the rules in $S_{\text{Comp}}^{\text{att}}$. The edges labeled by $S_{\text{Fw}}^{\text{att}}$ and $S_{\text{Recv}}^{\text{att}}$ in Figure 2.22 are located between the honest and attacker phases, and say that before stepping into the next phase, first, the tree is extended with a rule in $S_{\text{Fw}}^{\text{att}}$ and $S_{\text{Recv}}^{\text{att}}$, respectively. Let us call the subtree in Ph-A as PhA-tree. The root of Ph2-tree is a fact of form $\text{att}(l_{\text{att}}^{\text{rep}}, t_{\text{rep}})$ or $\text{att}(l_{\text{att}}^{\text{req}}, t_{\text{req}})$, where both $t_{\text{rep}}$ and $t_{\text{req}}$ are of form $(\text{head}; v_1; \ldots; [\text{List}]; \ldots; v_h)$. In my method, I attempt to find attacks with minimal steps. (I.) First of all, the algorithm examines whether the attack could be performed if the attacker forwards the reply unchanged. (II.) Otherwise, the algorithm examines what messages should the attacker intercept to be able to compose message $(\text{head}; v_1; \ldots; [\text{List}]; \ldots; v_h)$. The explanation of the pseudo-code for phase Ph-H is as follows (Table 1): This phase is defined by the function PhH($F_{\text{wm}}$), in which we assume that the derivation procedure has reached some $\text{wm}$-fact or the fact $\text{accept}([\text{List}_{\text{invalid}}])$, denoted by $F_{\text{wm}}$, and we continue with searching for a possible derivation of $F_{\text{wm}}$. The searching procedure is basically a Depth-First search.
At the beginning, if $F_{wm}$ is the fact $wm(t^p_{req\text{init}})$, where the pattern $t^p_{req\text{init}}$ represents the initial request sent by the source, then the deduction procedure terminated successfully, and the attack scenario $(T_{top}, M_{msg}, A)$ is returned (point 1). Otherwise, we search for a protocol rule $R$ that is resolvable with $F_{wm}$ and has not been applied to $F_{wm}$ before during the current verification procedure. If the $wm$-fact, $F_{wm}$, corresponds to a request message then the search is performed in the set of the request-rules, otherwise, we choose a rule $R$ in the set of reply-rules. The examination of the type of $F_{wm}$ is performed by the resolutions $R_{\text{req} \circ F_{wm}} F_{wm}$, and $R_{\text{reply} \circ F_{wm}} F_{wm}$. $F_{wm}$ has a request type (reply type) if the first (second) resolution is successful. In case we cannot find any such $R$, this means that the derivation got stuck, because we cannot reach the $\epsilon$ leaf in the derivation tree (point 2). The next step is to go upward in the tree to the father of $F_{wm}$, $F_{father}$, and search for another possible derivation path for it. There are two possibilities, $F_{father}$ is a $wm$-fact or a $att$-fact. In the first case, the deduction procedure is continued with searching for another derivation path for $F_{father}$, while in the latter case we step into the attacker phase Ph-A (points 3-4). If $F_{wm}$ does not have a father, i.e., $F_{wm} = \text{accept}([L_{\text{invalid}}])$, then not any derivation for $F_{wm}$ can be found. This means that no attack scenario is detected, and $(\emptyset, \emptyset, \emptyset)$ is returned (point 5).

In case there is a suitable $R$ that can be resolved with $F_{wm}$ (points 6-8), we perform the resolution, and get the children of $F_{wm}$ as result. Among these children, the $nbr$-facts, $wm$-facts or $att$-facts are added to the sets $nbr\text{facts}(F_{wm})$, $wm\text{facts}(F_{wm})$ or $att\text{facts}(F_{wm})$, respectively. If $T_{\text{invalid}} \subseteq T_{top}$ holds after updating $T_{top}$ with the $nbr$-facts in $nbr\text{facts}(F_{wm})$, then this derivation path will not lead to an attack, because at this point the route defined by $L_{\text{invalid}}$ becomes valid in the topology. Hence, we terminate this derivation path and choose another suitable $R$.

Points 9-13 consider the case when $T_{\text{invalid}} \not\subseteq T_{top}$, and we put the $wm$-facts and $att$-facts in the sets $wm\text{facts}(F_{wm})$ and $att\text{facts}(F_{wm})$ into $M_{msg}$ and $A$, respectively. These two sets record the messages exchanged by the honest nodes and the attacker nodes during the attack scenario, respectively. Finally, the whole procedure starts again, but now searching for a derivation of a child of $F_{wm}$. The pseudo-code of the attacker phase can be found in Table 2.

### Table 1: Pseudo-code of the honest phases ($PhH(F_{wm})$)

1. if ($F_{wm}$ is the fact $wm(t^p_{req\text{init}}})$ then return $(T_{top}, M_{msg}, A)$; else
   beginElse
     if $F_{wm}$ has a request type then return $S := S_{req} \cup S_{wm}$; goto point 2.
   endIf
   else if $F_{wm}$ has a reply type then return $S := S_{wm} \cup S_{rep}$; goto point 2.
2. Choose a rule $R \in S$ with which $F_{wm}$ can be extended & not applied to $F_{wm}$ before;
3. if ($\notin R \in S$ that has not been chosen before for extending $F_{wm}$) then
   4. if ($F_{wm}$ has a $wm$-fact typed father, $F_{\text{father}}^\text{att}$) then goto 1. with $F_{wm} := F_{\text{father}}^\text{att}$;
   5. else if ($F_{wm}$ has an $att$-fact typed father, $F_{\text{father}}^\text{att}$) then step into $Ph-A$ with $F_{\text{att}} := F_{\text{father}}^\text{att}$;
   6. else return $(\emptyset, \emptyset, \emptyset)$;
   beginElse
7. Let denote the set of $nbr$-facts among the children of $F_{wm}$ by $nbr\text{facts}(F_{wm})$;
8. Update $T_{\text{top}}$ with $nbr\text{facts}(F_{wm})$;
9. if ($T_{\text{invalid}} \subseteq T_{top}$) then remove $nbr\text{facts}(F_{wm})$ from $T_{top}$; goto 1.; else
   beginElse
10. Let denote the set of $wm$-facts among the children of $F_{wm}$ by $wm\text{facts}(F_{wm})$;
11. Update $T_{\text{top}}$ with $wm\text{facts}(F_{wm})$;
12. Remove $nbr$-facts in $nbr\text{facts}(F_{wm})$ from the tree;
13. Update $M_{msg}, A$ with the facts from $wm\text{facts}(F_{wm})$ and $att\text{facts}(F_{wm})$, respectively;
14. goto 1. with $F_{wm} :=$ a fact in $wm\text{facts}(F_{wm})$,
   or step into phase $Ph-A$ with an $att$-fact $F_{\text{att}}$ in $att\text{facts}(F_{wm})$;
   endElse
   endELSE
endELSE
endElse
Then, the deduction procedure is continued based on the type of \( t^p_{\text{att}} \). The function \( \text{FNotKeyedFunc}(F_{\text{att}}) \) is responsible for case when \( t^p_{\text{att}} \) is a constant, a node ID, a session ID, or a keyless crypto function (point a6). Function \( \text{FAttNotKeyedFunc}(F_{\text{att}}) \) handles the case when \( t^p_{\text{att}} \) is a keyed crypto function, and the key is owned by the attacker (point a8). If \( t^p_{\text{att}} \) is a keyless crypto function but the attacker does not own the key, then we proceed to examine how the attacker node can intercept or receive the request/reply message that contains \( t^p_{\text{att}} \) (points a9-a12). Function \( \text{REPREQcorrectplace}(F_{\text{att}}) \) analyzes the scenario when \( F_{\text{att}} \) is in a correct place within a request or a reply message, while \( \text{REPREQincorrectplace}(F_{\text{att}}) \) considers the case when \( t^p_{\text{att}} \) is located in an incorrect place (for example, a MAC is put into the place of a session ID).

In function \( \text{FNotKeyedFunc}(F_{\text{att}}) \), if \( F_{\text{att}} \) has more than one children in the current derivation tree, then all of these children are put into \( W \) (with performing deduplication steps). Thereafter, \( F_{\text{att}} \) is removed from \( W \) (to avoid deduction loop), and we go on with analyzing the following \( att \)-fact in \( W \) (points a6.2-3). If \( F_{\text{att}} \) does not have any child (i.e., \( t^p_{\text{att}} \) is a constant, a node ID, a session ID, or a keyless function such as a one-way hash function) (point a6.4). In this case, we check whether \( F_{\text{att}} \) or its only child is an element of the knowledge base of the attacker (i.e., whether \( F_{\text{att}} \) belongs to \( \epsilon \)) (point a6.5). If yes, then this deduction branch terminates successfully (point a6.6), otherwise, we reason about how the attacker can obtain the request or reply that contains \( t^p_{\text{att}} \) (point a6.7).

In function \( \text{FAttKeyedFunc}(F_{\text{att}}) \), where \( t^p_{\text{att}} \) is a keyed crypto function and the key is owned by the attacker, we reason about how the data part can be obtained. In particular, the data part is decomposed into smaller parts (point a8.1), then we update \( W \) with the resulted \( att \)-facts (point a8.2). Finally, \( F_{\text{att}} \) is removed from \( W \) (point a8.3).
 Function $\text{FNoTKeypFunc}(F_{\text{att}})$

a6.1. if ($\#\text{child}(F_{\text{att}}) > 1$) then 
  beginIF 
  a6.2. Put all the children of $F_{\text{att}}$ into $W$, and perform deduplications. 
  a6.3. Delete $F_{\text{att}}$ from $W$; $\text{Goto}$ point a2. in MAIN; 
  endIF else 
  a6.4. if ($F_{\text{att}}$ has no child OR $\#\text{child}(F_{\text{att}}) = 1$) then 
    beginIF 
    a6.5. if ($F_{\text{att}}$ OR the child of $F_{\text{att}} \in \text{Knowledge of the attacker}$) then 
      beginIF 
      a6.6. Delete $F_{\text{att}}$ from $W$; $\text{Goto}$ point a2. in MAIN; 
      endIF else $\text{Goto}$ point a9. in MAIN; 
    endIF 
  endIF 

 Function $\text{FRePreQcorrectplace}(F_{\text{att}})$

a8.1 Decomposing $\text{att}(W_{\text{Data}})$ with $A_{\text{comp}}$ yields some factset $W'$; 

/* Let $\text{att}(\text{Data}) \wedge \text{att}(W_{\text{Data}}, \text{key}) \rightarrow \text{att}(L_{\text{att}}, L_{\text{att}})$, where $F_{\text{att}} = \text{att}(L_{\text{att}}, L_{\text{att}})$ */ 

a8.2 Put $W'$ into $W$ and then eliminate fact duplication in $W$; 

a8.3 Delete $F_{\text{att}}$ from $W$; $\text{Goto}$ point a2. in MAIN; 

 Function $\text{FAttKeypFunc}(F_{\text{att}})$

a11.1 Extend $F_{\text{att}}^{\text{com}}$ with some rule $R, R \in S_{\text{att}}^{\text{Rec}}$, yielding a $\text{wm-fact} F_{\text{wm}}$, and step into $\text{PhH}(F_{\text{wm}})$; 

a11.2 if (In $\text{PhH}(F_{\text{wm}})$ the search does not get stuck, i.e., every leaf is $\epsilon$ & $T_{\text{invaid}} \not\subseteq T_{\text{top}}$) then 
  beginIF 
  a11.3 Decompose $F_{\text{att}}^{\text{com}}$ which yields some $W'$; 
  a11.4 $W = W - \{W' \cup F_{\text{att}}\}$; $\text{Goto}$ point a2. in MAIN; 
  endIF 

else if (we get stuck in $\text{PhH}(F_{\text{wm}})$, and step back into $\text{PhA}(F_{\text{att}}^{\text{max}})$ according to point a5. in $\text{PhH}(F_{\text{wm}})$) 
  beginELSE 
  a11.5 Decompose $F_{\text{att}}^{\text{com}}$ which yields some $W'$; 
  a11.6 while (3 rule $A_{\text{com}} \in \text{CONTAIN that has not been applied}$ do 
    beginWHILE 
    a11.7 Extend $\text{att}(W_{\text{Data}}) \in W'$ with $A_{\text{com}} \in \text{CONTAIN yielding att}(L_{\text{att}}, L_{\text{att}})$; 
    a11.8 Extend $\text{att}(L_{\text{att}}, L_{\text{att}})$ with $A_{\text{com}} \in \text{CONTAIN yielding att}(L_{\text{att}}, L_{\text{att}})$; 
    a11.9 if ($\text{att}(W_{\text{Data}})$ has NOT been examined before) then 
      beginIF 
      a11.10 Decompose $\text{att}(L_{\text{att}}, L_{\text{att}})$ which yields some $W''$; 
      a11.11 if ($F_{\text{att}} \in W''$) then 
        beginIF 
        a11.12 Extend $\text{att}(L_{\text{att}}, L_{\text{att}})$ with some $R, R \in S_{\text{att}}^{\text{Rec}}$, yielding a $\text{wm-fact} F_{\text{wm}}$, and step into $\text{PhH}(F_{\text{wm}})$; 
        a11.13 if (In $\text{PhH}(F_{\text{wm}})$ the search does not get stuck, i.e., every leaf is $\epsilon$ & $T_{\text{invaid}} \not\subseteq T_{\text{top}}$) then 
          beginIF 
          a11.14 $W = W - \{W'' \cup F_{\text{att}}\}$; 
          a11.15 Terminate Loop & $\text{Goto}$ point a2. in MAIN; 
          endIF 
        else if (the search steps back into $\text{PhA}(\text{att}(F_{\text{att}}^{\text{com}})$ according to point a5. in $\text{PhH}(F_{\text{wm}})$) 
          then $\text{goto}$ point a11.6. 
        endIF 
      endIF 
    endWHILE 
  endELSE 

a11.16 $\text{Goto}$ point a12. in MAIN; 

 Function $\text{FRePreQincorrectplace}(F_{\text{att}})$

a12.1 while ($\exists A_{\text{replace}} \in \text{REPLACE that has not been applied}$ do 
  beginWHILE 
  a12.2 Extend $F_{\text{att}}$, with $A_{\text{replace}}$ results in some $\text{att}(L_{\text{att}}, L_{\text{att}})$; 
  a12.3 if ($\text{att}(L_{\text{att}}, L_{\text{att}})$ has NOT been examined before) then 
    beginIF 
    a12.4 Extend $\text{att}(L_{\text{att}}, L_{\text{att}})$ with some rule $R, R \in S_{\text{att}}^{\text{Rec}}$, yielding a $\text{wm-fact} F_{\text{wm}}$, and step into $\text{PhH}(F_{\text{wm}})$; 
    a12.5 if (In $\text{PhH}(F_{\text{wm}})$ the search does not get stuck, i.e., every leaf is $\epsilon$ & $T_{\text{invaid}} \not\subseteq T_{\text{top}}$) then 
      beginIF 
      a12.6 Decompose $\text{att}(L_{\text{att}}, L_{\text{att}})$ which yields some $W''$; 
      a12.7 $W = W - \{W'' \cup F_{\text{att}}\}$; Terminate Loop & $\text{Goto}$ point a2. in MAIN; 
      endIF 
    else if (the search steps back into $\text{PhA}(\text{att}(F_{\text{att}}^{\text{com}})$ according to point a5. in $\text{PhH}(F_{\text{wm}})$) 
      then $\text{goto}$ point a12.1. 
    endIF 
  endWHILE 

a12.8 step into the honest phase $\text{PhH}(F_{\text{wm}})$, where $F_{\text{wm}}$ is a $\text{wm-fact}$ and the father of $F_{\text{att}}^{\text{com}}$. 

Function $\text{FRePreQcorrectplace}(F_{\text{att}})$ examines how the attacker node can intercept or receive the
Extending the algorithm for several attacker nodes

In this subsection, I discuss how sr-verif handles several attacker nodes, namely, how to extend the deduction algorithm above in order to handle more than one attacker node. The attacker rules in \( \mathcal{C}_{att} \) are specified by the variable \( l^p_{att} \) of attacker-ID type, e.g., \( A_{hash}^+ : att(l^p_{att}, x^p) \rightarrow att(l^p_{att}, h(x^p)) \), which define the knowledge and ability for the attacker node represented by \( l^p_{att} \). I assume that the ability of attacker nodes are the same, and they are distinguished based only on their IDs. When \([\text{List}\_\text{invalid}]\) contains no attacker ID, the set \( \mathcal{C}_{att} \) will not be instantiated with any attacker ID in \([\text{List}\_\text{invalid}]\), and in this case sr-verif assumes one attacker node represented by the variable \( l^p_{att} \). If \([\text{List}\_\text{invalid}]\) contains \( l^p_{att}, \ldots, l^p_{q\_att} \) attacker IDs, then sr-verif takes into account these \( q \) attackers during the verification.

During the backward deduction procedure, whenever a rule \( R \) from the set \( \mathcal{S}_{att}^{\text{new}} \) is applied in a resolution step, as result, \( l^p_{att} \) is bounded to some constant attacker ID, \( l^p_{att} \). Assume that \([\text{List}\_\text{invalid}]\) contains the attacker IDs \( l^p_{att}, \ldots, l^p_{q\_att} \). When a rule \( R \) in \( \mathcal{S}_{att}^{\text{old}} \) is used in a resolution during the backward reasoning, an att-fact, say \( att(l^p_{att}, (\text{head}^{req/rep}, v_1, \ldots, [\text{List}], \ldots, v_h)) \), is yielded and we step into the phase Ph-A. Then, \( l^p_{att} \) in \( att(l^p_{att}, (\text{head}^{req/rep}, v_1, \ldots, [\text{List}], \ldots, v_h)) \), is bounded to each of the \( l^p_{att}, \ldots, l^p_{q\_att} \). Intuitively, the attacker that received the message \((\text{head}^{req/rep}, v_1, \ldots, [\text{List}], \ldots, v_h)\) can be any of \( l^p_{att}, \ldots, l^p_{q\_att} \). The binding of \( l^p_{att} \) to the attacker IDs can be accomplished by decomposing the message, and resolve one of the resulting \( att(l^p_{att}, P_{att}), \ldots, att(l^p_{att}, l^p_{att}) \) with the fact \( att(l^p_{att}, l^p_{att}) \) in \( \mathcal{C}_{att} \).

Each time when \( l^p_{att} \) is bounded to \( l^p_{att} \), after a resolution step, an instance \( \mathcal{C}_{att} \) of \( \mathcal{C}_{att} \) is computed, where \( l^p_{att} \) is bounded to \( l^p_{att} \), in every rule. \( \mathcal{C}_{att} \) represents the knowledge and ability of node \( l^p_{att} \). Intuitively, during the whole deduction procedure \( \mathcal{C}_{att} \) is used to instantiate the knowledge and ability for the specific attackers with the IDs in \([\text{List}\_\text{invalid}]\). Basically, sr-verif starts with one attacker node, and continually increases the number of attacker nodes by inserting additional node IDs into the list \([\text{List}\_\text{invalid}]\). Hence, sr-verif covers the possible scenarios regarding the number and the locations of the attacker nodes. In addition, after \( l^p_{att} \) is bounded to \( l^p_{att} \), the new nbr-facts of the form \( nbr(l^p_{att}, l^p_{j}) \), in the set \( \mathcal{T}_{top} \), which is resulted from the last resolution step, become \( nbr(l^p_{att}, l^p_{j}) \). Namely, the new nbr-facts are instantiated by binding \( l^p_{att} \) to \( l^p_{att} \).
In case of q attacker nodes, sr-verif defines each set $W_1$, ..., $W_q$ for each attacker node $l^p_{att}$, ..., $l^v_{att}$, respectively. Each $W_i$ contains the $att$-facts, $att(l^i_{att}, tmstate)$, enqueued for examination and derivation. Because $l^p_{att}$ represents any attacker node, after binding it to $l^v_{att}$, the operation $l^v_{att}$ is the same as $l^p_{att}$ defined by $PhA(F_{root, att})$. During the backward deduction, whenever we get into the phase $PhA(F_{root, att})$, with $F_{root, att} = att(l^p_{att}, \text{head}_{\text{req/rep}}, v_1, \ldots, [\text{List}][\ldots, v_h])$, first, the leftmost attacker ID in $[\text{List}]$ is bounded to $l^v_{att}$. Then, we search for the derivation of this instance of $F_{root}$ in the same way as defined in the algorithm above. In case the derivation gets stuck, the algorithm returns to this point and bind the next attacker ID to $l^v_{att}$. Phase $PhA(F_{root, att})$ ends when the derivation of all the $att$-facts in all the q sets $W_1$, ..., $W_q$ finished, where each derivation can either gets stuck or terminates successfully.

Finally, to take in account the network topologies that consist of more than one attacker node, in the protocol rules (as well as the Figures 2.18, 2.19, 2.20) I change every $y_{\text{honprev}}$ and $y_{\text{honnext}}$ to $y_{\text{nxt}}$ and $y_{\text{prev}}$, respectively, allowing them to be bounded to attacker nodes’ ID as well.

### 2.8.10 Termination

In this subsection, I discuss how the verification algorithm ensures termination. First, I examine the possibility of an infinite loop during the searching procedure. Loop could either occur in the honest or attacker phase.

I assume on-demand source routing protocols, where the ID list is placed in the request and reply messages. I also assume that the routing protocol to be verified was designed “correctly”, such that it is loop free, and every honest node only handles the request (reply) with the same reply messages. I also assume that the routing protocol to be verified was designed “correctly”, allowing them to be bounded to attacker nodes’ ID as well.

In the verification procedure, I ensure this with the following two assumptions:

1. I examine only such invalid route $\text{List}_{\text{invalid}}$ in $\text{accept}([\text{List}_{\text{invalid}}])$, where the node IDs are pairwise different. The reason is that, in most cases, on-demand source routing protocols are defined/designed such that honest nodes will drop the request or reply if it contains the list with duplicated node IDs.

   The checking of duplicated IDs is the most basic protection measure against invalid route when we design a routing protocol. Note that I can extend my deduction algorithm for detecting loop during the protocol run, but this falls outside the focus of the dissertation, where I aim at detecting more critical weaknesses regarding the security issue. Moreover, the required steps for detecting loop can be protocol specific, depending on the particular contents of request/reply messages.

2. I assume that the routing protocol is specified such that each request includes the information about the node which sent it, while a reply message contains information about both the sender and the addressee (of course, this assumption is not necessarily valid to the messages sent by the attackers). This assumption is valid to all the examined representative on-demand source routing protocols DSR, SRP, Ariadne, endairA, where in the request message the last node ID in the list belongs to the sender node, while both the addressee and the sender are encoded in a reply message.

   Note that even in case the request/reply messages of a given protocol do not contain these information, we always can add them explicit without affecting the correctness of the protocol. For instance, if the protocol is defined such that the request is broadcast unchanged by the honest nodes, then in the corresponding protocol rule

   $\text{wm}(y^p_{\text{req}}) \land \text{nbr}(l^p_i, l^p_j) \rightarrow \text{wm}(y^p_{\text{req}})$

we cannot determine which $l^p_i$ is the sender of the message $y^p_{\text{req}}$. Therefore, in order for the deduction algorithm to operate properly, we have to add the ID of the sender into the $\text{wm}$-facts, for instance, $\text{wm}((l^p_i, y^p_{\text{req}}))$. Note that the attached ID, $l^p_i$, is not part of the request message, and only need for enabling the automated deduction. After attaching IDs into the requests, the rule above changed to $\text{wm}((l^p_i, y^p_{\text{req}})) \land \text{nbr}(l^p_i, l^p_j) \rightarrow \text{wm}((l^p_j, y^p_{\text{req}}))$. 

---

59
2. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WIRELESS AD-HOC ROUTING PROTOCOLS

Lemma 1. Besides the assumptions I provided above, in the honest phase \( Ph_H(F_{wm}) \), the deduction will not step into an infinite loop.

Proof. First of all, point 3 of \( Ph_H(F_{wm}) \) prevents the usage of the same rule \( R \) infinite amount of time for the a given \( F_{wm} \), by keeping track of the rules already used before. Hence, since the number of rules that can be resolvable with \( F_{wm} \) is finite, the deduction algorithm examines the derivation of a given \( F_{wm} \) only within a finite period of time.

In the second part of the proof, I will show that during the tree extending process (in the depth-first search manner) with consecutive resolution steps, we will never get into an infinite deduction loop. Formally, when searching for the derivation of some \( wm \)-fact \( F_{wm} \), the given deduction tree branch will not contain the same \( F_{wm} \) again and again, infinitely many times. Within Ph-\( H \) I further distinguish the request and reply directions in which the request and the reply messages are exchanged between honest nodes, respectively.

- During the request phase, the message exchanges among honest nodes are simulated by the protocol rules \( R^{req}_{2,3} \), \( R^{req}_{2,4} \) and \( R^{req}_{3} \). After each (backward) resolution step, \( R^{req}_{2,3} \circ_{F_{wm}} F_{wm} \), where \( F_{wm} = wm(t_{reqi}) \) and \( R^{req}_{2,3} = wm(y_{reqi}^p) \land \text{nbr}(l_i^p, l_p^t) \rightarrow wm(y_{reqi}^p) \land \text{nbr}(l_i^p, l_p^t) \) as result. This means that in order to make \( l_i^p \) be able to send the request \( t_{reqi}^p \), its neighbor node \( l_i^p \) should have sent the request \( t_{reqi}^p \). Based on the assumption that \( List_{invalid} \) contains finite number of different node IDs, and the protocol rules \( R^{req}_{2,3} \), \( R^{req}_{2,4} \) and \( R^{req}_{3} \) specify message exchanges between different nodes, it follows that the deduction procedure will terminate within finite number of resolution steps. The resolution steps will be performed constantly using \( R^{req}_{2,3} \) until we reach to the point when the initial request has been sent by the source node, where eventually the rules \( R^{req}_{2,4} \) and \( R^{req}_{3} \) are used.

- The situation is similar in case of the reply phase. Again, let the reply \( t_{rep}^{p, rep} \) that has been sent by node \( y_{i-1}^p \) to \( y_{prev}^{p, rep} \) includes the ID list \( \{ List, y_{prev}^{p, rep}, y_{prev, next}^{p, rep}, List \} \). The algorithm searches for the rules in \( S_{non} \) to extend the tree at \( wm(y_{prev, rep}^{p, rep}) \). Intuitively, we are reasoning about how and from whom the reply \( (y_{prev, rep}^{p, rep}) \) must be sent. After extending \( wm(y_{prev, rep}^{p, rep}) \) the fact \( wm(l_{i-1}^p, t_{rep}) \) is yielded, which means that the reply \( (y_{prev, rep}^{p, rep}) \) must be sent by node \( l_i^p \), who had to receive the reply \( l_i^p \). I recall that the ID at the beginning of the message represents the addressee of the reply. Due to the addressess of a reply is taken from \( \{ List_{invalid} \} \), the algorithm gets into an infinite loop only in case either the length of \( List_{invalid} \) is not finite or there are duplicated IDs in the list, which cannot happen by assumption. After applying the rules \( R^{3,1}, R^{4,4} \), the resolution steps will be performed consecutively using \( R^{3,12} \) (or \( R^{4,1} \) in case the network has only one intermediate node) until we reach to the point when the destination node sent back a reply after it received a request, which is modeled by the resolution steps with the rules \( R^{3,8} \) and \( R^{3,11} \).

To summarize, in both the request and reply phases, after performing each resolution step (i.e., tree extending step) we step from node to node in \( List_{invalid} \). Because the number of node IDs in \( List_{invalid} \) is finite, the honest phase will terminate within finite number of resolution steps.

I continue with showing that the attacker phase Ph-\( A(F_{root}) \) is free from infinite loop as well.

Lemma 2. During the search for a derivation of accept([\( List_{invalid} \)]) the algorithm does not get into an infinite deduction loop in the attacker phase Ph-\( A(F_{root}) \).

Proof. In attacker phases an infinite computation loop could happen when (i.) the attacker repeatedly performs some function \( f \) and its inverse counterparts \( f^{-1} \). For instance, the composition and decomposition rules \( A^{\text{comp}} \) and \( A^{\text{comp}} \) are performed iteratively in turn. However, in my method I prevent this by performing decomposition only, and instead of using composition rule to set up the required message I introduced the set of rules \( A^{\text{msg}} \), which derive the whole request or reply that contains the given (smaller) message part. (ii.) An another case which may cause loop is that the rule \( A^{\text{contain}} \) may be performed infinitely many times. However, this is not the case because each
application of $A_{\text{contain}}^\text{List}$ introduces a new node in the network, which contains only finite number of nodes. Formally, $A_{\text{contain}}^\text{List}$ is allowed only to applied up to the number of nodes in the network, which is assumed to be finite.

In $PhA(F^\text{root}_{\text{att}})$ we search for the derivation of the $\text{att}$-facts placed in $W$. The $\text{att}$-facts in $W$ represent the message parts of the requests and replies that are supposed to be sent, and its number is finite because the request and reply messages are composed of finite number of message elements. Because we perform deduplications after putting new facts into $W$, the size of $W$ is at most equal to the number of message parts of a request and a reply message.

In addition, I prevent deduction loop by also keeping track of the $\text{att}$-facts that are already examined during the current deduction procedure (point $a1$ of $PhA(F^\text{root}_{\text{att}})$). Further, whenever we get into the state where we need to derive the same $\text{att}$-fact again, we stop continuing this deduction branch. In point $a10$ of $PhA(F^\text{root}_{\text{att}})$, and points $a11.9$ and $a12.3$, we also keep track of the request/reply messages that we have examined before. Since the number of the possible requests/replies $t^\text{msg}_{\text{att}}$, $t^\text{broader}_{\text{att}}$ and $t^\text{replace}_{\text{att}}$, and the $\text{att}$-facts in $W$ are finite, the total number of resolutions performed in $PhA(F^\text{root}_{\text{att}})$ will be finite as well.

Finally, I show that during the whole deduction procedure, the occurrences of the honest and the attacker phases are finite. Namely, I prove that the algorithm steps from $PhA$ to $PhH$, and vice versa, only finite number of times. In the request phase, whenever we search for a derivation of a given $\text{wm}$-fact $F^\text{root}_{\text{wm}}$, we can step into an attacker phase by performing a resolution with one of the rules $R^\text{req}_{2,5}$, $R^\text{req}_{2,6}$, and $R^\text{req}_{2,7}$, yielding an $\text{att}$-fact $F^\text{root}_{\text{att}}$ (where $F^\text{root}_{\text{att}} = \text{att}(t^\text{req}_{\text{att}}, t^\text{req}_{\text{req}})$, for some request message $t^\text{req}_{\text{req}}$). According to the deduction steps defined in $PhA$, we have to search for the derivation of every message element in $t^\text{req}_{\text{req}}$, which is finite. In the worst case, we step into phase $PhA$ after points $a11.6$-$a11.7$ of function $\text{REQREPcorrectplace}$, when we search for the derivation of the fact $\text{att}(t^\text{broader}_{\text{att}}, t^\text{broader}_{\text{att}})$ within the phase $PhA$. Request message $t^\text{broader}_{\text{att}}$ contains the broader list than $\text{List}_{\text{invalid}}$, which we get by inserting new honest node IDs into $\text{List}_{\text{invalid}}$. Let the number of the node IDs in the ID list of $t^\text{broader}_{\text{att}}$ be $j$. In the honest phase (request phase), during get back from the destination to the source, we step into $PhA(F^\text{root}_{\text{att}})$ at most $j$ times (shown in the Figure 2.23). Each time, after getting into $PhA(F^\text{root}_{\text{att}})$ we can step into the honest phase $PhH(F^\text{root}_{\text{wm}})$, and in that honest phase we again can get into $PhA$, and so on. However, this circle cannot occur infinitely many times, because (i) in points $a1$-$a2$ of $PhA(F^\text{root}_{\text{att}})$, $F^\text{root}_{\text{att}} = \text{att}(t^\text{root}_{\text{att}}, (\text{head}^{\text{req}}; v_1; \ldots; [\text{List}]; \ldots; v_h))$, we search for the derivation of each element $\text{att}(t^\text{req}_{\text{att}}, (\text{head}^{\text{req}}); v_1), \ldots, \text{att}(t^\text{root}_{\text{att}}, [\text{List}]), \ldots, \text{att}(t^\text{root}_{\text{att}}, v_h)$ of the request, but only in case it has not been examined before within a session; (ii) the request contains finite parts of elements (i.e., $k$ and $[\text{List}]$ are finite). Hence, when we run out of the message elements that have not been examined before, the attacker phase will always get stuck at point $a2$, and the deduction procedure returns to phase $PhH(F^\text{root}_{\text{wm}})$ (point $a3$ of $PhA$), where we will step back to the source after at most $j$ resolution steps using the rules $R^\text{req}_{1,1}$, $R^\text{req}_{2,1}$, $R^\text{req}_{2,3}$. The situation is similar in case of the reply phase.
2.8.11 Correctness and completeness

In this subsection, I discuss about the correctness and the completeness which the proposed verification method provides.

**Correctness:** By correctness I mean whenever an attack is detected and returned it is really an attack. This property is declared in Lemma 3.

**Lemma 3.** Whenever an attack scenario \((T_{\text{top}}, M_{\text{msg}}, A)\) is returned for \([\text{List}_{\text{invalid}}]\), (i) the fact \(\text{accept}([\text{List}_{\text{invalid}}])\) has been derived and (ii) \(T_{\text{invalid}} \notin T_{\text{top}}\).

**Proof.** According to the algorithm, the attack scenario is returned only when every leaf in the derivation tree of \(\text{accept}([\text{List}_{\text{invalid}}])\) is \(\epsilon\), and \(T_{\text{invalid}} \notin T_{\text{top}}\) holds whenever the current topology, \(T_{\text{top}}\), is updated with new edges (i.e., additional nbr-facts are added to \(T_{\text{invalid}}\)).

**Completeness:** In general case, when an arbitrary source routing protocol is assumed, \(sr\)-verif is not complete. This means that when the verification procedure ends and no attack scenario is returned, there may be an attack in case of unexamined \(\text{List}_{\text{invalid}}\). This is due to the following assumptions:

- Although \(sr\)-verif does not assume any specific topology, the length of the invalid route, \(\text{List}_{\text{invalid}}\), for which the deduction algorithm attempts to find an attack, is finite. Hence, in general case, when the algorithm returns \((\emptyset, \emptyset, \emptyset)\) for a given \(\text{List}_{\text{invalid}}\) of length \(k\), then it means that the attackers cannot achieve that the source accepts an invalid route of length \(k\).

- I restrict the number of applying the same rule \(A^\text{contain}_{\text{List}}\) in CONTAIN. Intuitively, I limit the number of new node IDs (i.e., \(l^p[\_]\), which is differ from the IDs in \(\text{List}_{\text{invalid}}\)) that the attacker can insert into the ID list in request and reply messages. In case of the examined representative source routing protocols (such as Ariadne) an attack can be found after only one-time application of \(A^\text{contain}_{\text{List}}\). This represents the attack scenario against Ariadne, in which the attacker obtain the required signatures and hashes for constructing the incorrect request, by waiting for a (broader) request message that contains these elements.

Otherwise, the deduction algorithm is exhaustive: The set of protocol rules takes into account every possible scenario. The set of rules also covers all the possible scenarios of the message exchanges between the honest and the attacker nodes. Hence, the request and the reply phases can be simulated by the series of resolutions defined in the honest phase \(PhH(F_{\text{am}})\). In the attacker phase, \(PhA\), the attacker can only perform an attack if it obtains all the message elements of the request and reply messages which are needed for a successful attack. This is ensured in the first two points of \(PhA(F_{\text{att}})\). To examine how the attacker could obtain each part of the required request/reply, we examine (i) whether the attacker can compute the particular element based only on its knowledge and ability; (ii) if not, then how can it receives/intercepts from a honest node or an another attacker node. The attacker attempts to insert message parts into an incorrect place within a request or a reply. Finally, the attacker tries to insert additional node IDs into the ID list, in order to see whether the request/reply that contains this new ID list contains the required message element. The rationale behind the inserting of IDs to the ID list is that in most source routing protocols the content of the ID list determine the structure of the whole request/reply message. Based on the protocol specification, the attacker will not try to change/forge the information that will be verified by the honest nodes, and the verification will fail for sure. This can be ensured by using type interference during resolutions.

2.8.12 Complexity

I assume that the length of \([\text{List}_{\text{invalid}}]\) in the fact \(\text{accept}([\text{List}_{\text{invalid}}])\) is \(k\). I define the complexity of the proposed algorithm by the number of resolution steps that needed to be performed. The following cases are distinguished and examined:
• **The complexity of the request phase without stepping into the attacker phase:** In \( \text{PhH}(F_{\text{wm}}) \), checking the type of \( F_{\text{wm}} \) takes two resolution steps in the worst case (point 1). One resolution is for checking if \( F_{\text{wm}} \) is a request, and an another if it is a reply. Then, in point 2 \( F_{\text{wm}} \) is extended with one of the rules \( R_{2,3}^{\text{req}}, R_{2,2}^{\text{eq}} \) and \( R_{3,1}^{\text{req}} \), which takes three resolutions. This requires at most three resolution steps. Moreover, in points 7-8, \( \text{nbr} \)-facts are resulted after each resolution step. Eliminating one \( \text{nbr} \)-fact takes one resolution step \( \text{nbr}(l_{\text{att}}, l_{\text{bro}}) \) or \( \text{nbr}(y_{\text{att}}, y_{\text{bro}}) \). Let the number of \( \text{nbr} \)-facts in rules \( R_{2,2}^{\text{eq}} \) and \( R_{3,1}^{\text{eq}} \) be \( \text{nbr}_{2,1}^{\text{req}} \) and \( \text{nbr}_{2,3}^{\text{req}} \), which yield \( \max(\text{nbr}_{2,1}^{\text{req}}, \text{nbr}_{2,3}^{\text{req}}) \) resolution steps.

In the request phase, we can get back from the destination to the source node by continually performing the resolution steps with the rule \( R_{2,3}^{\text{req}} \), until we reach the source node, where the two rules \( R_{2,1}^{\text{req}} \) and \( R_{3,1}^{\text{req}} \) are applied, in this order (shown in Figure 2.24). In the worst case, the number of resolutions is equal to the number of ID list in the request message \( t_{\text{bro}}^{\text{att}} \) in the fact \( \text{att}(p_{\text{att}}, i_{\text{bro}}^{\text{att}}) \). Let the number of IDs in this ID list be \( r \). Hence, in the request phase, \( (2 + 3 \times \max(\text{nbr}_{2,1}^{\text{req}}, \text{nbr}_{2,3}^{\text{req}})) \times \max(r, k) \) resolution steps are required.

![Figure 2.24: The backward reasoning is based on consecutive resolution steps. After performing each resolution step with a given rule, we step back from one node to its neighbor node. On the left side, we consider the case when there is more than one intermediate node, while on the right side, the case of one honest intermediate node is illustrated.](image)

• **The complexity of the reply phase without stepping into the attacker phase:** Based on the similar reasoning, in the reply phase we have to search in the set \( S_{\text{rep}}^{\text{rep}}, S_{\text{rep}}^{\text{rep}} = \{ R_{1,1}^{\text{rep}}, R_{2,1}^{\text{rep}}, R_{2,4}^{\text{rep}}, R_{2,12}^{\text{rep}}, R_{3,1}^{\text{rep}} \} \), which costs 6 resolution steps. In the reply phase, we can get back from the source to the destination by applying \( R_{1,1}^{\text{rep}} \) first, followed by using the rule \( R_{2,4}^{\text{rep}} \). Then, the resolution steps between the resulted fact from the previous resolutions and \( R_{2,12}^{\text{rep}} \) are constantly performed. Finally, the rules \( R_{2,1}^{\text{rep}} \) and \( R_{3,1}^{\text{rep}} \) are used to get into the request phase. In case of one intermediate node, rule \( R_{2,1}^{\text{rep}} \) is applied after using \( R_{1,1}^{\text{rep}} \) and before \( R_{3,1}^{\text{rep}} \). Therefore, in the reply phase, at most \( (2 + 6 \times \max(\text{nbr}_{2,1}^{\text{rep}}, \text{nbr}_{2,3}^{\text{rep}}, \text{nbr}_{2,12}^{\text{rep}}, \text{nbr}_{2,3}^{\text{rep}})) \times \max(r, k) \) resolution steps are required.

• **The complexity of the attacker phase, \( \text{PhA}(F_{\text{root}}) \):** The resolution steps required in one attacker phase depends on the number of message elements in the requests and replies supposed to be sent by the attacker (i.e., the size of set \( W \)). For each \( \text{att} \)-fact \( f_{\text{att}} \) in \( W \), we search for the rule in the set of the attacker’s computation ability, \( \mathcal{A}_{\text{att}}, \) which can be resolvable with \( f_{\text{att}} \). This takes \( |\mathcal{A}_{\text{att}}| \) number of resolution steps, where \( |\mathcal{A}_{\text{att}}| \) denotes the size of \( \mathcal{A}_{\text{att}} \).

In function \( \text{FNotKeyedFunc}(F_{\text{att}}) \), the case (\#\( \text{childs}(F_{\text{att}}) > 1 \)) yields \( \#\text{childs}(F_{\text{att}}) + 1 \) resolution steps. In case (\#\( \text{childs}(F_{\text{att}}) = 1 \)) or \( F_{\text{att}} \) has no child, we need to examine whether \( F_{\text{att}} \) or its child is in the attacker’s knowledge set, \( \mathcal{K}_{\text{att}}, \) which takes \( |\mathcal{K}_{\text{att}}| \) resolution steps. Hence, the worst case complexity of \( \text{FNotKeyedFunc} \) is: Complex(\( \text{FNotKeyedFunc} \)) = \( \max(\#\text{childs}(F_{\text{att}}) + 1, |\mathcal{K}_{\text{att}}|) \).

In point \( a7 \) of \( \text{PhA}(F_{\text{root}}) \), we have to examine if the key is in the union set \( I_{\text{att}}^{\text{key}} \cup I_{\text{att}}^{p_{\text{key}}} \), which requires \( |I_{\text{att}}^{\text{key}}| + |I_{\text{att}}^{p_{\text{key}}}| \) resolution steps.
In $F\text{AttKeyedFunc}(F_{\text{att}})$, the attacker rule $A^{\text{comp}}$ is used to extend $\text{att}(l^p_{\text{att}}, \text{Data})$, which takes one resolution step. Deleting $F_{\text{att}}$ in point a8.3 also takes one resolution step.

Point a9 of $PhA(F^\text{root}_{\text{att}})$ costs one resolution step. Let the set that stores the already examined $\text{att}$-facts for $l^\text{msg}_\text{att}$, $l^\text{broader}_\text{att}$ and $l^\text{replace}_\text{att}$ messages be $W_{\text{msg}}$, $W_{\text{broader}}$, and $W_{\text{replace}}$, respectively.

In point a10, we have to examine if $F_{\text{att}}$ has been examined before, which requires $|W_{\text{msg}}|$ resolutions.

In $\text{REQREPcorrectplace}$, points a11.1 and a11.5 take $|S^\text{recv}_{\text{att}}|$ and one resolution steps, respectively. In the while cycle, points a11.7 and a11.8 require two resolution steps, points a11.10 and a11.11 take $|W_{\text{broader}}|$ and $|W''|$ resolution steps, respectively. Finally, point a11.12 costs $|S^\text{recv}_{\text{att}}|$ steps. In total, the complexity of the function is:

$$\text{Complex}(\text{REQREPcorrectplace}) = |\text{CONTAIN}| \times (|S^\text{recv}_{\text{att}}| + |W_{\text{broader}}| + |W''|) + |S^\text{recv}_{\text{att}}| + 1.$$ 

Within the while construct of $\text{REQREPincorrectplace}$, points a12.2, a12.3 and a12.4 require one, $|W_{\text{replace}}|$ and $|S^\text{recv}_{\text{att}}|$ resolution steps, respectively. Hence,

$$\text{Complex}(\text{REQREPincorrectplace}) = |\text{REPLACE}| \times (|W_{\text{replace}}| + |S^\text{recv}_{\text{att}}| + 1).$$

To summarize, the worst-case complexity of phase $PhA(F^\text{root}_{\text{att}})$ is:

$$\text{Complex}(\text{PhA}) = 1 + |W| \times (|W_{\text{att}}| + |C_{\text{att}}| + \text{MAX}(\text{Complex}(F\text{NotKeyedFunc}), \text{Complex}(F\text{AttKeyedFunc}), |I^\text{key}_{\text{att}}| + |I^\text{key}_{\text{att}}| + \text{MAX}(\text{Complex}(\text{REQREPcorrectplace}), \text{Complex}(\text{REQREPincorrectplace}))).$$

The worst-case complexity of the backward deduction algorithm, for a given $[\text{List}_{\text{invalid}}]$, can be upper bounded by

$$\text{const} \times q \times \text{Complex}(\text{PhA})^{2 \times \text{max}(k,r)},$$

for some $\text{const}$ which specifies the resolution steps required in the honest phases before and after each attacker phase (which is a linear function of $\text{max}(k,r)$), and $q$ represents the number of the attacker nodes.

The complexity of the proposed algorithm in practise: Although in the worst case, the complexity of $sr$-verif is the exponential function of the length of the ID list $\text{List}_{\text{invalid}}$, in practise, it is very effective in case of well-known on-demand source routing protocols. In case of well-known routing protocols, an attack scenario can be found with $[\text{List}_{\text{invalid}}]$ containing not more than three node IDs. An attack scenario against the DSR protocol is found when $[\text{List}_{\text{invalid}}] = [l^p_1]$ is examined; an attack scenario against the SRP protocol is found at the point when the verification tool is examining the case in which $[\text{List}_{\text{invalid}}] = [l^p_1, l^p_2]$, and an attack scenario against the Ariadne protocol is detected in case $[\text{List}_{\text{invalid}}] = [l^p_1, l^p_2, l^p_{\text{att}}]$. Finally, the analyzed covert channel attack against the endairA protocol (discussed in Section 2.7.4) can be detected based on the invalid list $[l^p_{\text{att}}, l^p_{\text{att}}]$, where $l^p_{\text{att}}$ and $l^p_{\text{att2}}$ are the IDs of two different attacker nodes.

In practise, the complexity of the attacker phases is not large because the message elements in the $\text{att}$-fact $F_{\text{att}}$ corresponding to a latter $PhA(F^\text{root}_{\text{att}})$ phase is the subset of the message elements in the upper level attacker phases and the first $PhA(F^\text{root}_{\text{att}})$ phase. More specifically, the $\text{att}$-facts in the set $W$ will not increase exponentially, because the duplicated facts are eliminated.

Finally, despite considering an arbitrary topology and a strong attacker node, my proposed approach is more effective than the approach in [7], which handles specific topology. The main advantage of my approach is that to verify a routing protocol it does not have to examine exhaustively all the topologies, which is required in [7]. In [7] the authors exhaustively check $2^{\frac{n(n-1)}{2}}$ or $2^{n(n-1)}$ topologies for $n$ nodes. This is a bad approach because they also check a large number of equivalent topologies. The SPIN model checker is applied for each topology to detect attacks.
2.8.13 Implementation

I have developed the first prototype version of the software tool based on the proposed theoretical foundations. The tool is implemented in the JAVA programming language, and succeeded in finding attack scenario in case of the (the first version of) SRP and Ariadne protocols. For all these examples, the total time for finding derivation trees is less than 1 second on a laptop with Core 2 Duo 2800Mhz processor. The extension of the prototype implementation to handle several attacker nodes will be addressed in the future work.

2.9 Summary

In this chapter, I argued that designing secure ad-hoc network routing protocols requires a systematic approach which minimizes the number of mistakes made during the design. To this end, I proposed a variant of process algebra called the sr-calculus, which provides expressive syntax and semantics for analyzing at the same time (i.) cryptographic primitives and operations, (ii.) the nature of broadcast communication, and (iii.) the specification of node’s neighborhood in wireless medium, which are required for verifying secure routing protocols. I proposed a systematic and exhaustive proof technique, called BDSR, for analyzing routing protocols with the sr-calculus.

In addition, I proposed a fully automatic verification method, called sr-verif, for secured ad-hoc network routing protocols, which is based on logic and a backward reachability approach. My method has a clear syntax and semantics for modeling secure routing protocols, and handles arbitrary network topologies. Finally, my method can be used to verify the security of source routing protocols when the network includes several attacker nodes, who can cooperate with each other, and run several parallel sessions of the protocol. My publications related to this topic are [Th05, 2010], [Th06, 2011], [Th07, 2011], [Th08, 2012].
Chapter 3

Formal and automated security verification of WSN transport protocols

3.1 Introduction

Wireless Sensor Networks (WSNs) consist of a large number of resource constrained sensor nodes and a few more powerful base stations. The sensors collect various types of data from the environment and send those data to the base stations using multi-hop wireless communications. Some typical applications that require the use of a transport protocol for ensuring reliable delivery and congestion control are: assured delivery of high-priority events to sinks; reliable control and management of sensor networks; remotely programming/retasking sensor nodes over-the-air; and multimedia sensor networks [6], where the sensors capture and transmit high-rate data with some QoS requirements.

It is widely accepted that transport protocols used in wired networks (e.g., the well-known TCP) are not applicable in WSNs, because they perform poorly in a wireless environment and they are not optimized for energy consumption. Therefore, a number of transport protocols specifically designed for WSNs have been proposed in the literature (see, e.g., [72] for a survey), where many of WSN transport protocols require intermediate nodes on a path between the source and destination to cache data fragments until they can be sure that they have been delivered [26], [9], [67], [70]. The main design criteria that those transport protocols try to meet are reliability and energy efficiency. However, despite the fact that WSNs are often envisioned to operate in hostile environments, existing transport protocols for WSNs do not address security issues at all and, as a consequence, they ensure reliability and energy efficiency only in a benign environment where no intentional attack takes place [16].

Broadly speaking, attacks against WSN transport protocols can be attacks against reliability and energy depleting attacks. An attack against reliability is considered to be successful if the loss of a data packet (or packet fragment) remains undetected. In case of energy depleting attacks, the goal of the attacker is to force the sensor nodes to perform energy intensive operations, in order to deplete their batteries. An example would be when the attacker coerces some sensor nodes to unnecessarily retransmit data packets (or packet fragments).

Numerous transport protocols have been proposed specifically designed for applications of wireless sensor networks (WSN), requiring particularly reliable delivery and congestion control (e.g., multimedia sensor networks [72]). Three of the latest protocols are the Distributed Transport for WSNs (DTSN) [47], and its secured version, the Secure Distributed Transport Protocol for WSNs (SDTP) [17], and my proposed protocol, the Secure Distributed Transport Protocol for WSNs based on hash-chain and Merkle-tree (SDTP⁺) [Th11 , 2013]. In DTSN, SDTP and SDTP⁺, the intermediate nodes can cache the packets with some probability and retransmit them upon request, providing reliable transmission, energy efficiency and distributed functionality.
3. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WSN TRANSPORT PROTOCOLS

Systematic mathematical and automated methods are needed for finding the weaknesses in the mentioned protocols and similar protocols, however, this is a very hard task due to their complexity. This served as a motivation and challenge for my work, to which I proposed solutions. In my dissertation, I address the problem of formal and automated security verification of WSN transport protocols, which typically consist of the following behavioral characteristics: (b1) storing data packets in the buffer of intermediate sensor nodes; (b2) probabilistic and timed behavior; (b3) performing cryptographic operations such as one-way hashing, computing message authentication codes (MACs), and so on. I propose a formal and an automated verification method, based on the application of a process algebra and a model-checking framework, respectively. For demonstration purposes, I apply the proposed methods for specifying and verifying the security of the DTSN, the SDTP, and the SDTP+ protocols, which are representative in the sense that DTSN involve the first two behavioral characteristics (b1-b2), while SDTP and SDTP+ cover all of the three points (b1-b3). Specifically, the main contributions of this chapter of the dissertation are the following:

- I propose a probabilistic timed calculus, called cryptprob\_time, for cryptographic protocols [Th13, 2013], [Th12, 2013]. To the best of my knowledge, this is the first of its kind in the sense that it combines the following three features: (i.) it supports formal syntax and semantics for cryptographic primitives and operations; (ii.) it supports time constructs similar to the concept of timed automata that enables us to verify real time systems; (iii.) it also includes the syntax and semantics of probabilistic constructs for analyzing systems that perform probabilistic behavior. The basic concept of cryptprob\_time is inspired by the previous works [27], [31], [24] proposing solutions separately for each of the three discussed points. In particular, cryptprob\_time is derived from the applied \( \pi \)-calculus [27], which defines an expressive syntax and semantics supporting cryptographic primitives to analyze security protocols; a probabilistic extension of the applied \( \pi \)-calculus [31]; and a process calculus for timed automata proposed in [24].

Note that, although in this chapter of the dissertation the proposed cryptprob\_time calculus is used for analyzing WSN transport protocols, it is also suitable for reasoning about other systems that include cryptographic operations, as well as timed and probabilistic behavior.

- I propose a new secured transport protocol for WSNs, called SDTP+ [Th11, 2013]. Using cryptprob\_time, I specify the behavior of the previously proposed DTSN and SDTP protocols, as well as the behavior of my proposed SDTP+. I propose the novel definition of weak probabilistic timed bisimilarity and use it to prove the weaknesses of DTSN and SDTP, as well as the security of SDTP and SDTP+ against some attacks.

- I provide an approach for the automatic security verification of the DTSN and SDTP protocols with the PAT process analysis toolkit [68], which is a powerful general-purpose model checking framework. To the best of my knowledge PAT has not been used for this purpose before, however, in this dissertation I show that the power of PAT can be used to check some interesting security properties defined for these systems/protocols [Th13, 2013], [Th12, 2013].

3.2 Previously proposed transport protocols for WSNs

In this section, I provide an overview of two recently proposed transport protocols for WSNs, the DTSN [47] and the SDTP [17] protocols.

3.2.1 DTSN - Distributed Transport for Sensor Networks

DTSN [47] is a reliable transport protocol developed for sensor networks where intermediate nodes between the source and the destination of a data flow cache data packets in a probabilistic manner such that they can retransmit them upon request. The main advantages of DTSN compared to a transport protocol that uses a fully end-to-end retransmission mechanism is that it allows
3.2. Previously proposed transport protocols for WSNs

intermediate nodes to cache and retransmit data packets, hence, the average number of hops a retransmitted data packet must travel is smaller than the length of the route between the source and the destination. Intermediate nodes do not store all packets but only store packets with some probability $p$, which makes it more efficient. Note that in the case of a fully end-to-end reliability mechanism, where only the source is allowed to retransmit lost data packets, retransmitted data packets always travel through the entire route from the source to the destination. Thus, DTSN improves the energy efficiency of the network compared to a transport protocol that uses a fully end-to-end retransmission mechanism.

DTSN uses special packets to control caching and retransmissions. More specifically, there are three types of such control packets: Explicit Acknowledgement Requests (EARs), Positive Acknowledgements (ACKs), and Negative Acknowledgements (NACKs). The source sends an EAR packet after the transmission of a certain number of data packets, or when its output buffer becomes full, or when the application has not requested the transmission of any data during a predefined timeout period or due to the expiration of the EAR timer (EAR timer).

The activity timer and the EAR timer are launched by the source for ensuring that a session will finish in a finite period of time. The activity timer is launched when the source starts to handle the first data packet in a session, and it is reset when a new packet is stored, or when an ACK or a NACK has been handled by the source. When the activity timer has expired, depending on the number of unconfirmed data packets, the session will be terminated or reset. The EAR timer is launched whenever an EAR packet or a data packet with the EAR bit set is sent.

An EAR may take the form of a bit flag piggybacked on the last data packet or an independent control packet. An EAR is also sent by an intermediate node or the source after retransmission of a series of data packets, piggybacked on the last retransmitted data packet [47]. Upon receipt of an EAR packet the destination sends an ACK or a NACK packet, depending on the existence of gaps in the received data packet stream. An ACK refers to a data packet sequence number $n$, and it should be interpreted such that all data packets with sequence number smaller than or equal to $n$ were received by the destination. A NACK refers to a base sequence number $n$ and it also contains a bitmap, in which each bit represents a different sequence number starting from the base sequence number $n$. A NACK should be interpreted such that all data packets with sequence number smaller than or equal to $n$ were received by the destination and the data packets corresponding to the set bits in the bitmap are missing.

Within a session, data packets are sequentially numbered. The Acknowledgement Window (AW) is defined as the number of data packets that the source transmits before generating and sending an EAR. The output buffer at the sender works as a sliding window, which can span more than one AW. Its size depends on the specific scenario, namely on the memory constraints of individual nodes.

In DTSN, besides the source, intermediate nodes also process ACK and NACK packets. When an ACK packet with sequence number $n$ is received by an intermediate node, it deletes all data packets with sequence number smaller than or equal to $n$ from its cache and passes the ACK packet on to the next node on the route towards the source. When a NACK packet with base sequence number $n$ is received by an intermediate node, it deletes all data packets with sequence number smaller than or equal to $n$ from its cache and, in addition, it retransmits those missing data packets that are indicated in the NACK packet and stored in the cache of the intermediate node. The bits that correspond to the retransmitted data packets are cleared in the NACK packet, which is then passed on to the next node on the route towards the source. If all bits are cleared in the NACK, then the NACK packet essentially becomes an ACK referring to the base sequence number, and it is processed accordingly. In addition, the intermediate node sets the EAR flag in the last retransmitted data packet. The source manages its cache and retransmissions in the same way as the intermediate nodes, without passing on any ACK and NACK packets.

Reasoning about the security of DTSN

Upon receiving an ACK packet, intermediate nodes delete from their cache the stored messages whose sequence number is less than or equal to the sequence number in the ACK packet, be-
cause the intermediate nodes believe that acknowledged packets have been delivered successfully. Therefore, an attacker may cause permanent loss of some data packets by forging or altering ACK packets. This may put the reliability service provided by the protocol in danger. Moreover, an attacker can trigger unnecessary retransmission of the corresponding data packets by either setting bits in the bit map of the NACK packets or forging/altering NACK packets. Any unnecessary retransmission can lead to energy consumption and interference. Note that, unnecessary retransmissions do not directly harm the reliability, but it is clear that such inefficiency is still undesirable.

The destination sends ACK or NACK packets upon reception of an EAR. Therefore, attacks aiming at replaying or forging EAR information, where the attacker always sets the EAR flag to 0 or 1, can have a harmful effect. Always setting the EAR flag to 0 prevents the destination from sending an ACK or NACK packet, while always setting it to 1 forces the destination send control packets unnecessarily.

3.2.2 SDTP - A Secure Distributed Transport Protocol for WSNs

SDTP [17] is a security extension of DTSN aiming at patching the security holes in DTSN. SDTP ensures that an intermediate node can verify if an acknowledgment or negative acknowledgment information has really been issued by the destination, if and only if the intermediate node actually has in its cache the data packet referred to by the ACK or NACK. Forged control information can propagate in the network, but only until it hits an intermediate node that cached the corresponding data packet; this node can detect the forgery and drop the forged control packet.

In particular, the security solution of SDTP works as follows [17]: each data packet is extended with an ACK MAC and a NACK MAC, which are computed over the whole packet with two different keys, an ACK key (K_{ACK}) and a NACK key (K_{NACK}). Both keys are known only to the source and the destination and are specific to the data packet; hence, these keys are referred to as per-packet keys.

When the destination receives a data packet, it can check the authenticity and integrity of each received data packet by verifying the two MAC values. Upon receipt of an EAR packet, the destination sends an ACK or a NACK packet, depending on the gaps in the received data buffer. If the destination sends an ACK referring to a data packet with sequence number n, the destination reveals (included in the ACK packet) the corresponding ACK key; similarly, when it wants to signal that this data packet is missing, the destination reveals the corresponding NACK key by including it in the NACK packet. Any intermediate node that stores the packets in question can verify if the ACK or NACK message it receives is authentic by checking if the appropriate MAC in the stored data packet verifies correctly with the ACK key included in the ACK packet. In case of successful verification, the intermediate node deletes the corresponding data packets (whose sequence number is smaller than or equal to n) from its cache.

When an ACK packet is received by an intermediate node or the source, the node first checks if it has the corresponding data packet. If not, then the ACK packet is simply passed on to the next node towards the source. Otherwise, the node uses the ACK key obtained from the ACK packet to verify the ACK MAC value in the data packet. If this verification is successful, then the data packet can be deleted from the cache, and the ACK packet is passed on to the next node towards the source. If the verification of the MAC is not successful, then the ACK packet is silently dropped.

When a NACK packet is received by an intermediate node or the source, the node processes the acknowledgement part of the NACK packet as described above. In addition, it also checks if it has any of the data packets that correspond to the set bits in the bitmap of the NACK packet. If it does not have any of those data packets, it passes on the NACK without modification. Otherwise, for each data packet that it has and that is marked as missing in the NACK packet, it verifies the NACK MAC of the data packet with the corresponding NACK key obtained from the NACK packet. If this verification is successful, then the data packet is scheduled for re-transmission, the corresponding bit in the NACK packet is cleared, and the NACK key is removed from the NACK packet. After these modifications, the NACK packet is passed on to the next node towards the source.
3.3. SDTP$^+$ - A Secure Distributed Transport Protocol for WSNs based on Hash-chain and Merkle-tree

The $ACK$ and $NACK$ key generation and management in SDTP is as follows: The source and the destination share a secret which I call the session master key, and I denote it by $K$. From this, both the source and destination derive an $ACK$ master key $K_{ACK}$ and a $NACK$ master key $K_{NACK}$ for a given session as follows:

$$K_{ACK} = \text{PRF}(K; \text{"ACK master key"}; \text{SessionID})$$
$$K_{NACK} = \text{PRF}(K; \text{"NACK master key"}; \text{SessionID})$$

where $\text{PRF}$ is a pseudo-random function [25], and SessionID is a session identifier.

SDTP assumes a pre-established shared secret value, such as a node key shared by the node and the base station, which can be configured manually in the node before its deployment. Denoting the shared secret by $S$, the session master key $K$ is then derived as follows:

$$K = \text{PRF}(S; \text{"session master key"}; \text{SessionID})$$

The $ACK$ key $K_{ACK}^{(n)}$ and the $NACK$ key $K_{NACK}^{(n)}$ for the $n$-th packet (i.e., whose sequence number is $n$) are computed as follows:

$$K_{ACK}^{(n)} = \text{PRF}(K_{ACK}; \text{"per packet ACK key"}; n)$$
$$K_{NACK}^{(n)} = \text{PRF}(K_{NACK}; \text{"per packet NACK key"}; n)$$

Note that both the source and the destination can compute all these keys as they both possess the session master key $K$. Moreover, PRF is a one-way function, therefore, when the $ACK$ and $NACK$ keys are revealed, the master keys cannot be computed from them, and consequently, the yet unrevealed $ACK$ and $NACK$ keys remain secrets too.

Reasoning about the security of SDTP

The rationality behind this security solution is that the shared secret $S$ is never leaked, and hence, only the source and the destination can produce the right $ACK$ and $NACK$ master keys and per-packet keys. Since the source never reveals these keys, the intermediate node can be sure that the control information has been sent by the destination. In addition, because the per-packet keys are computed by a one-way function, when the $ACK$ and $NACK$ keys are revealed, the master keys cannot be computed from them; hence, the yet unrevealed $ACK$ and $NACK$ keys cannot be derived. These issues give the protocol designers an impression that SDTP is secure, however, I will formally prove that SDTP is still vulnerable and show a tricky attack against it.

The main security weakness of the SDTP protocol is that the intermediate nodes store the received data packets without any verification. Intermediate nodes do not verify the origin and the authenticity of the data packets or the $ACK$ and the $NACK$ messages, namely, they cannot be sure whether the data packets that they stored were sent by the source node, and the control messages were really sent by the destination. Indeed, the security solution of SDTP only enables intermediate nodes to verify the matching or correspondence of the stored packets and the revealed $ACK/NACK$ keys. Hence, SDTP can be vulnerable in case of more than one attacker node (compromised node) who can cooperate, which I will confirm during my analysis.

3.3 SDTP$^+$ - A Secure Distributed Transport Protocol for WSNs based on Hash-chain and Merkle-tree

I propose SDTP$^+$ [Th11, 2013], in order to patch the security weaknesses can be found in DTSN and SDTP. SDTP$^+$ aims at enhancing the authentication and integrity protection of control packets, and is based on an efficient application of asymmetric key crypto and authentication values, which are new compared to SDTP. The security mechanism of SDTP$^+$ is based on the application of Merkle-tree [49] and hash chain [23], which have been used for designing different security protocols such as Castor [28], a scalable secure protocol for ad-hoc networks, and Ariadne.
My contribution is applying Merkle-tree and hash chain in a new context. The general idea of SDTP+ is the following: two types of “per-packet” authentication values are used, ACK and NACK authentication values. The ACK authentication value is used to verify the ACK packet by any intermediate node and the source, whilst the NACK authentication value is used to verify the NACK packet by any intermediate node and the source. The ACK authentication value is an element of a hash chain [23], whilst the NACK authentication value is a leaf and its corresponding sibling nodes along the path from the leaf to the root in a Merkle-tree [49]. Each data packet is extended with one Message Authentication Code (MAC) value (the MAC function is HMAC), instead of two MACs as in SDTP.

SDTP+ adopt the notion and notations of the pre-shared secret $S$, ACK, and NACK master secrets $K_{ACK}$, $K_{NACK}$, which are defined and computed in exactly the same way as in SDTP (Section 3.2.2). However, in SDTP+ the generation and management of the per-packet keys $K_{ACK}^n$, $K_{NACK}^n$ is based on the application of hash-chain and Merkle-trees, which is different from SDTP.

### 3.3.1 The ACK Authentication Values

The ACK authentication values are defined to verify the authenticity and the origin of ACK messages. The number of data packets that the source wants to send in a given session, denoted by $m$, is assumed to be available. At the beginning of each session, the source generates the ACK master secret $K_{ACK}$ (like in Section 3.2.2) and calculates a hash chain of size $(m+1)$ times, which is illustrated in Figure 3.1. Each element of the calculated hash-chain represents a per packet ACK authentication value as follows: $K_{ACK}^{m}, K_{ACK}^{m-1}, ..., K_{ACK}^{1}, K_{ACK}^{0}$, where $K_{ACK}^{i} = h(K_{ACK}^{i+1})$ and $h$ is a one-way hash function. The value $K_{ACK}^{0}$ is the root of the hash-chain, and $K_{ACK}^{i}$ represents the ACK authentication value corresponding to the packet with sequence number $i$. When the destination wants to acknowledge the successful delivery of the $i$-th data packet, it reveals the corresponding $K_{ACK}^{i}$ in the ACK packet.

![Figure 3.1](image.png)

**Figure 3.1:** The element $K_{ACK}^{i}$, $i \in \{1, \ldots, m\}$, of the hash-chain is used for authenticating the packet with the sequence number $i$. The root of the hash-chain, $K_{ACK}^{0}$, which we get after hashing $(m+1)$ times the ACK master key $K_{ACK}$. This root is sent to every intermediate node in the open session packet, which is digitally signed by the source.

### 3.3.2 The NACK Authentication Values

For authenticating the NACK packets, SDTP+ applies a Merkle-tree (also known as hash-tree), which is illustrated in Figure 3.2. When a session has started, the source computes, in the same way like in Section 3.2.2, the NACK per-packet keys $K_{NACK}^n$ for each packet to be sent in a given session. Afterwards, these NACK per-packet keys are hashed and assigned to the leaves of the Merkle-tree: $K_{NACK}^n = h(K_{NACK}^{n})$. The internal nodes of the Merkle-tree are computed as the hash of the (ordered) concatenation of its children. The root of the Merkle-tree, $H(h_s, S_s)$, is sent by the source to intermediate nodes in the same open session packet that includes the root of the hash-chain. For each $K_{NACK}^j$, $j \in \{j_1, \ldots, j_m\}$, the so called sibling values $S_{i_1}, \ldots, S_{i_t}$, for some
3.3. SDTP – A Secure Distributed Transport Protocol for WSNs based on Hash-chain and Merkle-tree

$t$, are defined such that the root of the Merkle-tree can be computed from them. For instance, the sibling values of $K^{(j)}_{NACK}$ are $K^{(j)}_{NACK}$, $S_1$, $\ldots$, $S_p$. From these values $H(h_p, S_p)$ can be computed.

![Figure 3.2: The structure of Merkle-tree used in SDTP+. Each internal node is computed as the hash of the ordered concatenation of its children. The root of the tree, $H(h_p, S_p)$, is sent out by the source.]

### 3.3.3 The operation of the source

When a session is opened, first, the source computes the ACK and NACK master keys $K_{ACK}$ and $K_{NACK}$, respectively. Then, the source calculates the hash-chain and the Merkle-tree for the session. Afterwards, the source sends an open session message with the following parameters: the roots of the hash chain ($K^{(0)}_{ACK}$) and of the Merkle-tree ($H(h_p, S_p)$), the length of the hash chain ($m+1$), the session $sID$, the source and destination IDs. Before sending the open session packet, the source digitally signs it to prevent the attackers from sending fake open session packets.

After receiving an ACK message corresponding to the session open packet, from the destination, the source starts to send data packets. Each data packet is extended with the MAC, computed over the whole packet (except for the EAR and RTX flags), using the shared secret between the source and the destination.

When the source node receives an ACK packet that includes the ACK authentication value ($K^{(i)}_{ACK}$), corresponding to the packet of sequence number $i$, it hashes iteratively the ACK authentication value $i$ times. If the result is equal to the stored root hash value $K^{(0)}_{ACK}$, then the ACK packet is accepted and the source removes all the packets of sequence number smaller than or equal to $i$ from its cache. Otherwise, the ACK packet is ignored and dropped.

Assume that the source node receives a NACK packet,

$$NACK = (i, [j_1, \ldots, jk], K_{ACK}, [K^{(j_1)}_{NACK}, \ldots, K^{(j_k)}_{NACK}], S_{i_1}^{j_1}, \ldots, S_{i_k}^{j_k}, S_{j_1}^{j_1}, \ldots, S_{j_k}^{j_k})$$

where $i$ and $[j_1, \ldots, jk]$ are the sequence numbers of the acknowledged packets, and the list of packets to be re-transmitted. $K^{(i)}_{ACK}$ is the ACK authentication value for packet $i$, and $K^{(j)}_{NACK}$ are the NACK authentication values for the packets $j_1, \ldots, jk$. $S_{i_1}^{j_1}, \ldots, S_{i_k}^{j_k}$ are the sibling values corresponding to $K^{(j)}_{NACK}$, and $S_{j_1}^{j_1}, \ldots, S_{j_k}^{j_k}$ are the values corresponding to $K^{(j)}_{NACK}$, required for the source to compute the root of the Merkle-tree.

The source first checks the ACK authentication value and performs the same steps as described above for ACK authentication. Then the source continues with verifying the NACK authentication values. For each $j$ in $[j_1, \ldots, jk]$, the source re-computes the root of the Merkle-tree based on the received NACK authentication values, $K^{(j)}_{NACK}$, $S_{i}^{j_1}, \ldots, S_{i}^{j_k}$, and compares the root of the resultant tree with the stored root. If the two roots are equal, then the NACK packet is accepted and the source retransmits the required packets.
3.3.4 The operation of the destination

When the destination node receives an open-session packet sent by the source, it verifies the signature computed on the packet. Upon success, the destination starts to generate the ACK and NACK master keys, the hash-chain, and the Merkle-tree. Finally, the destination sends an ACK packet to the source for acknowledging the delivery of the open session packet. Upon receiving a data packet with sequence number \( i \), first, the destination checks MAC using the secret shared between the source and the destination. Upon success, the destination delivers the packet to the upper layer. Otherwise, the packet is ignored and dropped. Upon the receipt of a packet with a set EAR flag, the destination sends an ACK or a NACK packet depending on the existence of gaps in the received data packet stream. The ACK packet that refers to sequence number \( i \) is composed of the pair \((i, K_{ACK}^{(i)})\). Similarly, the NACK packet with base sequence number \( i \) is extended with the ACK authentication value \( K_{ACK}^{(i)} \), and if the destination wants to request for re-transmission of some packet \( j \), then it also includes the corresponding NACK authentication values \( K_{NACK}^{(j)}, S_1^j, \ldots, S_q^j \) in the NACK packet.

3.3.5 The operation of the intermediate nodes

Upon receipt of an open session packet and the corresponding ACK packet, an intermediate node verifies signature computed on the packet, and in case of success, it stores the root values of the hash chain and the Merkle-tree, the session ID, sID, and forwards the packet towards the destination. Otherwise, an intermediate node changes its probability to store packets in the current session to zero. Upon receipt of a data packet, an intermediate node stores with probability \( p \) the data packet and forwards the data packet towards the destination.

When an intermediate node receives an ACK packet, \((i, K_{ACK}^{(i)})\), it verifies the authenticity and the origin of the ACK message by hashing \( K_{ACK}^{(i)} \) \( i \) times, and comparing the result with the stored root value of the hash chain. If the two values are equal (i.e., \( K_{ACK}^{(i)} = h^i(K_{ACK}^{(i)}) \)), then all the stored packets with the sequence number less than or equal to \( i \) are deleted. Afterwards, the intermediate node passes on the ACK packet towards the source. Otherwise, the ACK packet is ignored and dropped. When intermediate nodes receive a NACK packet that refers to the sequence numbers \( i \) and \([j_1, \ldots, j_k]\), they perform verification steps in the same way as the source. Namely, based on the ACK authentication value, the root of the hash-chain is re-computed, while based on the received NACK authentication values the root of the Merkle-tree is re-generated. The resultant roots are compared with the stored roots, and in case of equality, the cache entries are deleted based on \( i \) and the stored data packets are re-transmitted based on \([j_1, \ldots, j_k]\). Afterwards, the intermediate node removes from \([j_1, \ldots, j_k]\] the sequence numbers of the packets it has already re-transmitted, and forwards the NACK with the modified list towards the source.

3.3.6 Reasoning about the security of SDTP+

The main difference between SDTP and SDTP+ is that intermediate nodes verify the authenticity of the roots of the hash-chain and the Merkle-tree. Hence, based on the revealed ACK and NACK authentication values, the stored roots can only be computed if they are revealed by the destination.

In case of modifying the ack value \( n \) to a larger \( m \) in ACK packets, to be successful, beside changing the ACK value the attacker has to include a correct ACK authentication value \( K_{ACK}^{(m)} \). Note that only \( K_{ACK}^{(0)}, \ldots, K_{ACK}^{(n)} \) are revealed so far by the destination, and the ACK authentication values are the elements of a hash-chain. Therefore, computing \( K_{ACK}^{(m)} \) based on \( K_{ACK}^{(0)}, \ldots, K_{ACK}^{(n)} \) is very hard because the hash function used in generating the hash-chain is one-way.

As for the case of NACK packets forgery, injecting valid NACK packets by either creating a whole NACK packet or modifying some sequence numbers of the packets to be re-transmitted. To be successful an attacker has to include a valid NACK authentication value (NACK authentication values and their siblings) into the NACK packet. Specifically, the attackers have to be able to
compute the leaves of the Merkle-tree, however, computing the leaves based on the upper level hash values is hard because the hash function used to generate the Merkle-tree is one-way.

The attackers can forge the roots of the hash-chain and the Merkle-trees by modifying the open session packet, however, the open session packet is digitally signed by the source. Hence, the attackers have to forge the signature, which is also very hard.

3.4 crypt: The calculus for cryptographic protocols

crypt is the base calculus for specifying and analyzing cryptographic protocols, without supporting timed and probabilistic systems. crypt can be seen as a modified variant of the applied π-calculus [27], designed for analyzing security protocols, and proving their security properties in a convenient way. My goal is to extend crypt with time and probabilistic modeling elements adapting the concept of timed and probabilistic automata, and to do this, I need to modify the applied π-calculus in some points. Namely, I replace process replication with recursive process invocation; I add definition for positive integers, and comparison rules for them; I also define syntax for cache/buffer entry. Finally, note that I give the name crypt for the base calculus, instead of the name modified π-calculus, because I want a straightforward naming convention for the corresponding probabilistic and timed extensions of the base calculus.

3.4.1 Syntax and semantics

I assume an infinite set of names \( \mathcal{N} \) and variables \( \mathcal{V} \), where \( \mathcal{N} \cap \mathcal{V} = \emptyset \). Further, I define a set of distinguished variables \( \mathcal{E} \) that model the cache entries for of entities that store data. In the set \( \mathcal{N} \), I distinguish channel names, and other kind of data. Channel names are denoted by \( c_i \) with different indices such that \( c_i \neq c_j, i \neq j \). The set of non-negative integers is denoted by \( \mathcal{I} \), and its elements range over \( int_i \) with different indices that are corresponding to the numbers 0, 1, 2, etc.

Further, let the remaining data be denoted by \( m_i, n_i, k_i \). The variables are denoted by \( x_i, y_i, z_i \), and the cache entries by \( e_i \) with different indices. The names and variables with different indices are different. Let \( \mathcal{F} \) be the set of function symbols. To verify security protocols, in my case the function symbols capture the cryptographic primitives such as hash, encryption, MAC function. Finally, I assume the type system of the terms as in the applied π-calculus.

I define a set of terms as

\[
t ::= c_i \mid \text{int}_i \mid n_i, m_i, k_i \mid x_i, y_i, z_i \mid e_i \mid \emptyset \mid f(t_1, \ldots, t_k).
\]

where \( | \) represents “or”. In particular, a term can be the following:

- \( c_i \) models a communication channel between honest parties. Channels can be hidden (private), hence, messages sent on them cannot be overheard by the attackers, or they can be public channels;
- \( n_i, m_i, k_i \) are names and are used to model some data;
- \( x_i, y_i, z_i \) are variables that can represent any term, that is, any term can be bounded to variables. Similarly as in case of the applied π-calculus [27];
- \( e_i \) is a cache entry; the unique name \( \emptyset \) represents the empty content of a cache entry.
- Finally, \( f \) is a function with arity \( k \) and is used to construct terms and to model cryptographic primitives, and messages. Complex messages are modeled by the function \( \text{tuple} \) with \( k \) terms: \( \text{tuple}(t_1, \ldots, t_k) \), which I abbreviate as \( (t_1, \ldots, t_k) \). The function symbol with arity zero is a constant;
3. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WSN TRANSPORT PROTOCOLS

- \textit{int}_i\textit{ ranges over special functions for modeling non-negative integers. Formally, let 0 be the base element of set \mathcal{I}, and is formally defined as the function named by 0. Each further integer is defined as a constructor function named by 1, 2, etc. Let the function \textit{inc}(\textit{int}_i) be the function that increases the integer \textit{int}_i by one. Numbers 1, 2, \ldots are modeled by functions \textit{inc}(0), \textit{inc}(1), \ldots, respectively. The relation between these integers is defined by \textit{int}_i < \textit{inc}(\textit{int}_i) and \textit{int}_i = \textit{int}_i;}

The internal operation of communication entities in the system is modeled by \textit{processes}. Processes can be specified with the following syntax, and inductive definition:

\[
P, Q, R ::= \text{Processes} \\
\tau(t).P \mid c(x).P \mid (P|Q) \mid P[\ ]Q \mid \nu n.P \mid I(y_1, \ldots, y_n) \mid [t_i = t_j]P \text{ else } Q \mid [\textit{int}_i \geq \textit{int}_j]P \text{ else } Q \mid [\textit{int}_i > \textit{int}_j]P \text{ else } Q \mid [t_i = t_j]P \mid [\textit{int}_i \geq \textit{int}_j]P \mid [\textit{int}_i > \textit{int}_j]P \mid \text{n} \mid \text{let } (x = t) \text{ in } P \mid \text{let } (e = t) \text{ in } P.
\]

The meaning of each plain process is as follows:

- The process \tau(t).P represents the sending of message \( t \) on channel \( c \), followed by the execution of \( P \). Process \( c(x).P \) represents the receiving of some message, which is bounded to \( x \) in \( P \).

- In the composition \( P | Q \), processes \( P \) and \( Q \) run in parallel. Each may interact with the other on channels known to both, or with the outside world, independently from each other. For example, the communication between the sending process \( \tau(t).P \) and receiving process \( c(x).P \) can be described as the parallel composition \( \tau(t).P | c(x).Q \).

- A choice \( P | [] Q \) can behave either as \( P \) or \( Q \) depending on the first visible/invisible action of \( P \) and \( Q \). If the first action of \( P \) is enabled but the first action of \( Q \)'s is not then \( P \) is chosen, and vice versa. In case both actions are enabled the behavior is the same as a non-deterministic choice.

- A restriction \( \nu n.P \) is a process that makes a new, private (restricted) name \( n \), and then behaves as \( P \). The scope of \( n \) is restricted to \( P \), and is available only for the process within its scope. A private channel \( c \) restricted to \( P \) is defined by \( \nu c.P \), which does not allow for the attackers to eavesdrop on the channel.

- A typical way of specifying infinite behavior is by using parametric recursive definitions, like in the \( \pi \)-calculus [50]. Here \( I(y_1, \ldots, y_n) \) is an identifier (or invocation) of arity \( n \). I assume that every such identifier has a unique, possibly recursive, definition \( I(x_1, \ldots, x_n) \text{ def } P \) where the \( x_i \)'s are pairwise distinct. The intuition is that \( I(y_1, \ldots, y_n) \) behaves as \( P \) with each \( x_i \) replaced by \( y_i \), respectively.

- In processes \( [t_i = t_j]P \text{ else } Q \); \( [\textit{int}_i \geq \textit{int}_j]P \text{ else } Q \); and \( [\textit{int}_i > \textit{int}_j]P \text{ else } Q \); if \((t_i = t_j)\), \((\textit{int}_i \geq \textit{int}_j)\), and \((\textit{int}_i > \textit{int}_j)\), respectively, then process \( P \) is “activated”, else they behave as \( Q \). When \( Q \) is the \text{nil} process, the else branch is simply removed from the processes.

- The process \text{n} does nothing, and is used to model the termination of a process behavior.

- Finally, \text{let } (x = t) \text{ in } P \text{ (or let } (e = t) \text{ in } P \text{)} means that every occurrence of \( x \) (or \( e \)) in \( P \) is bounded to \( t \).

I adapt the notion of \textit{environment}, defined in many process algebras, which is used to model the attacker(s) who can obtain the (publicly) exchanged messages, and can modify, as well as send them. Every message \( t \), which is output on a public channel by the processes, is captured by the
environment. Moreover, I adopt the notation of the extended process and active substitution in
the applied π-calculus [27] to model the information the attacker(s) (or the environment) is getting
to know during the system run. The definition of the extended process is as follows:

\[
A, B, C ::= P \mid (A|B) \mid \nu n.A \mid \nu x.A \mid \{t/x\}
\]

- \(P\) is a plain network I already discussed above.
- \(A|B\) is a parallel composition of two extended process.
- \(\nu n.A\) is a restriction of the name \(n\) to \(A\).
- \(\nu x.A\) is a restriction of the variable \(x\) to \(A\).
- \(\{t/x\}\) means that the binding of \(t\) to \(x\), denoted by \(\{t/x\}\), is applied to any process that
is in parallel composition with \(\{t/x\}\). Intuitively, the binding applies to any process that comes into contact with it. To restrict the binding \(\{t/x\}\) to a process \(P\), I use the variable restriction \(\nu x\) over \(\{t/x\}|P\), namely, \(\nu x. (\{t/x\}|P)\). Using this, the equivalent definition of
process \(\tau(t).P\) can be given by \(\nu x.(\tau(x).P | \{t/x\})\). Active substitutions are always assumed to be cycle-free.

I define \(fv(A)\), \(bv(A)\), \(fn(A)\), and \(bn(A)\) for the sets of free and bound variables and free
and bound names of \(A\), respectively. These sets are defined as follow:

\[
fv(\{t/x\}) \overset{\text{def}}{=} fv(t) \cup \{x\}, fn(\{t/x\}) \overset{\text{def}}{=} fn(t); \ bv(\{t/x\}) \overset{\text{def}}{=} 0, bn(\{t/x\}) \overset{\text{def}}{=} bn(t)
\]

The concept of bound and free values is similar to local and global scope in programming
languages. The scope of names and variables are delimited by binders \(c(x)\) (i.e., input) and \(\nu n\) or \(\nu x\) (i.e., restriction). The set of bound names \(bn(A)\) contains every name \(n\) which is under the restriction \(\nu n\) inside \(A\). The set of bound variables \(bv(A)\) consists of all those variables \(x\) occurring in \(A\) that are bound by restriction \(\nu x\) or input \(c(x)\). Further, I define the set of free names and the set of free variables. The set of free names in \(A\), denoted by \(fn(A)\), consists of those names \(n\) occurring in \(A\) that are not restricted names. The set of free variables \(fv(A)\) contains the variables \(x\) occurring in \(A\) which are not restricted variables \(\nu x\) or input variable \(c(x)\). A plain process \(P\) is closed if it contains no free variable. An extended process is closed when every variable \(x\) is either bound or defined by an active substitution.

As in the applied π-calculus, a frame \(\varphi\) is an extended process built up from the \texttt{nil}
process and active substitutions of the form \(\{t/x\}\) by parallel composition and restrictions. Formally, the frame \(\varphi(A)\) of the extended process, \(A = \nu n_1 ... n_k(\{t_1/x_1\} | ... | \{t_n/x_n\} | P)\), is \(\nu n_1 ... n_k(\{t_1/x_1\} | ... | \{t_n/x_n\})\). The domain of the frame \(\varphi(A)\) (denoted by \textit{dom}(A)) is the set \(\{x_1, ..., x_n\}\).

Intuitively, the frame \(\varphi(A)\) accounts for the static knowledge exposed by \(A\) to its environment,
but not for dynamic behavior. The frame allows access to terms which the environment cannot
construct. For instance, after the term \(t\) (not available for the environment) are output in \(P\)
resulting in \(P' | \{t/x\}\), \(t\) becomes available for the environment. Finally, let \(\sigma\) range over substitutions
(i.e., variable bindings). I write \(\sigma t\) for the result of applying \(\sigma\) to the variables in \(t\).
3.5 \textit{crypt}^\text{prob}_\text{time}: Extending \textit{crypt} with timed and probabilistic syntax and semantics

I propose a time and probabilistic extension to \textit{crypt}, denoted by \textit{crypt}^\text{prob}_\text{time}. My proposed calculus is tailored for the verification of security protocols, especially for verifying protocols that need to cache data, such as transport protocols for wireless sensor networks. This is a new probabilistic timed calculus for cryptographic protocols, and to the best of my knowledge, the first of its kind. The design methodology of \textit{crypt}^\text{prob}_\text{time} is based on the terminology proposed in previous works, it can be seen as the modification and extension of them, and contains some novelties. Note that with \textit{crypt}^\text{prob}_\text{time}, my purpose is to develop a formal proof method for probabilistic timed cryptographic protocols, and the question of how can an automated verification method based on \textit{crypt}^\text{prob}_\text{time} be designed is left for the future. In the dissertation, I used the PAT process analysis toolkit [68] for automating the verification, instead of designing an automatic method based on \textit{crypt}^\text{prob}_\text{time}.

Namely, the timed extension of \textit{crypt} is based on the timed calculus proposed in [44], [24], and it is also based on the syntax and semantics of timed automata. The probabilistic extension is inspired by the syntax and semantics of the probabilistic extension of the applied π-calculus proposed in [31], and the probabilistic automata in [24]. The main difference between my work and the related methods is that I focus on extending \textit{crypt}, which is different from the calculus used in those works. In addition, I combine both timed and probabilistic elements at the same time. Finally, I also propose a new definition called \textit{weak probabilistic timed bisimilarity} for proving the existence of the attacks against security protocols.

The concept of \textit{crypt}^\text{prob}_\text{time} is based on the concept of probabilistic timed automata, hence, the correctness of \textit{crypt}^\text{prob}_\text{time} comes from the correctness of the automata because the semantics of \textit{crypt}^\text{prob}_\text{time} is equivalent to the semantics of the probabilistic timed automata, and I show that each process in \textit{crypt}^\text{prob}_\text{time} has an associated probabilistic timed automaton.

First of all, I give an overview of the basic time concepts, and provide some notations related to clocks and time constructs, borrowed from the concept of timed automata. Assume a set \( C \) of nonnegative real valued variables called clocks. A clock valuation over \( C \) is a mapping \( v: C \rightarrow \mathbb{R}^0 \) assigning nonnegative real values to clocks. For a time value \( v + d \) denote the clock valuation such that \((v + d)(x_c) = v(x_c) + d\), for each clock \( x_c \in C \).

The set \( \Phi(C) \) of clock constraints is generated by the following grammar:

\[
\phi ::= \text{true} \mid \text{false} \mid x_c \sim N \mid \phi_1 \land \phi_2 \mid \neg \phi
\]

where \( \phi \) ranges over \( \Phi(C) \), \( x_c \in C \), \( N \) is a natural number, \( \sim \in \{\lt, \leq, \gt, \geq\} \). I write \( v \models \phi \) when the valuation \( v \) satisfies the constraint \( \phi \). Formally, \( v \models \text{true}; v \models x_c \sim N \) iff \( v(x_c) \sim N \); \( v \models \phi_1 \land \phi_2 \) iff \( v \models \phi_1 \) and \( v \models \phi_2 \).

### 3.5.1 Formal syntax of \textit{crypt}^\text{prob}_\text{time}

In the following, I turn to define probabilistic timed processes for \textit{crypt}^\text{prob}_\text{time}:

\[
A_{pt} ::= A \mid \alpha^* \leftarrow \pi \ A_{pt} \mid \phi \leftarrow A_{pt} \mid \phi \rightarrow A_{pt} \mid \| C_R \| A_{pt} \mid A_{pt}^1 \mid A_{pt}^2 \mid X_{pt}
\]

I will discuss the meaning of \textit{crypt}^\text{prob}_\text{time} processes by showing the connection between the modeling elements of a probabilistic timed automaton and \textit{crypt}^\text{prob}_\text{time}. For this purpose, I recall the definition of probabilistic timed automaton [44]: a probabilistic timed automaton \( \text{Aut} \) is defined by the tuple \((\mathcal{L}, \phi_0, \sum, \mathcal{C}, \partial, \kappa, \mathbb{E}, \Pi)\), where
3.5. crypt$^{\text{prob}}$: Extending crypt with timed and probabilistic syntax and semantics

- $\mathcal{L}$ is a finite set of locations and $q_0$ is the initial location;
- $\sum$ is a set of actions that range over $\text{act}$;
- $\mathcal{C}$ is a finite set of clocks;
- $\delta: \mathcal{L} \mapsto \Phi(\mathcal{C})$ is a function that assigns location to a formula, called a location invariant, that must hold at a given location;
- $\kappa: \mathcal{L} \mapsto 2^\mathcal{C}$ is the set of clock resets to be performed at the given locations;
- $\mathbb{E} \subseteq \mathcal{L} \times \sum \times \Phi(\mathcal{C}) \times \mathcal{B} \times \mathcal{L}$ is the set of edges. I write $q \xrightarrow{act,\phi} q'$ when $(q, act, \phi, B, q') \in \mathbb{E}$, where $act$, $\phi$ are the action and the time constraint defined on the edge, and $B$ is the set of the clocks to be reset at $q'$.
- $\Pi = \{\pi_1, \ldots, \pi_n\}$ is a finite set of probability distributions. Each $\pi_i$ is a function $\pi_i: \mathbb{E} \mapsto [0,1]$ for any $i = \{1, \ldots, n\}$, where $\pi_i(g)$ is the probability of an edge $g$ according to distribution $\pi_i$, and the sum of the edges from a given location $q$ is 1.

Let us denote the set of processes in crypt$^{\text{prob}}$, by $A^{\text{prob}}$, and let $A^{\pi}_p$ range over processes in crypt$^{\text{time}}$. In crypt$^{\text{prob}}$, each probabilistic timed process $A^{\pi}_p$ corresponds to a location $q$ in an automaton, such that there is an initial process $A^{\pi}_p$ for location $q_0$. The set of actions $\sum$ corresponds to the set of actions known in crypt. The set of clocks to be reset at a given location $q$, $\mathcal{C}(q)$, is defined by the corresponding crypt$^{\text{prob}}$ process $||C_R||A_q^{\pi}$. The clock invariant at the location $q$ corresponds to the process $\phi \triangleright A^{\pi}_p$, and the edge guard can be defined by $\phi \triangleq A^{\pi}_p$. More specifically,

- $A_p$ can be an extended process $A$ without any time construct.
- $A^\ast \prec_q A_p$ performs $A^\ast$ as the first (not timed) action with the distribution $\pi$, at any time, and then it behaves like $A_p$. Note that $A^\ast$ can be $\nu x.\bar{x}(x), \bar{c}(u), c(t)$, and the silent action $\tau$. For instance, if $A_p$ is $c(t).P$, where $P$ is the plain process in crypt, then $A^\ast$ is $c(t)$. Let $A^{\pi}_p$ be the process that we get after performing action $A^\ast$ in $A_p$. Process $A^\ast \prec_q A_p$ corresponds to the automaton edge $q \xrightarrow{\alpha^\ast,\text{true}} q'$, where $A^\ast \prec_q A_p$ and $A'_p$ corresponds to locations $q$ and $q'$, respectively.
- $\phi \triangleq A_p$ represents a time guard of an action, and says that the first action $A^\ast$ of $A_p$ is performed if the guard (time constraint) $\phi$ holds. This process intends to model the edge $q \xrightarrow{\alpha^\ast,\phi} q'$ in the automaton syntax, where $A_p$ and $A'_p$ correspond to $q$ and $q'$, respectively, such that $A'_p$ is the process resulted after performing action $A^\ast$ in $A_p$. In the transition, the action $A^\ast$ is performed according to the distribution $\pi$. When an action has a time guard $\phi$ it means that the action can be performed at any time.
- $\phi \triangleright A_p$ represents a clock invariant over $A_p$. This process corresponds to the location invariant in an automaton. Like in timed automaton, this means that the system cannot “stay” in process $A_p$ once time constraint $\phi$ becomes invalid. If it cannot move from this process via any transition, then it is a deadlock situation. Invariants can be used to model timeout.

In the timed process $||C_R||A_p$, first, the clocks in the set $C_R$ are reset and then it behaves like $A_p$ with the reset clock values.

- $A^1_p \downarrow A^2_p$ and $A^1_p \uparrow A^2_p$ describe the first-action choice, and the parallel composition of two processes, respectively. Process $A^1_p \downarrow_p A^2_p$ behaves like $A^1_p$ with probability $p$, and it behaves as $A^2_p$ with $(1-p)$. $A^1_p \uparrow A^2_p$ corresponds to a location $q$ from which two edges start, and they are chosen based on the first enable action of $A^1_p$ and $A^2_p$. For parallel composition, I define $A^1_p \| A^2_p$ as a location, instead of the parallel composition of two automata. Process $A^1_p \downarrow_p A^2_p$ corresponds to a location $q$ from which two edges start: $q \xrightarrow{\alpha,\phi} q_1$ and $q \xrightarrow{\alpha,\phi} q_2$, where $q_1$ and $q_2$ correspond to $A^1_p$ and $A^2_p$, respectively.
Definition 12. I extend the definition of free and bound variables in Section 3.4 with the set of clock variables. The set of free variables and bound variables of $A_{pt}$, denoted by $\text{fvc}(A_{pt})$ and $\text{bvc}(A_{pt})$, respectively, are as follows:

- $fvc(\phi \hookrightarrow A_{pt}) = \text{clock}(\phi) \cup fvc(A_{pt})$: Edge guards contain free clock variables.
- $fvc(\phi \triangleright A_{pt}) = \text{clock}(\phi) \cup fvc(A_{pt})$: Invariant contains free clock variables.
- $\text{bvc}([C_R][A_{pt}]) = bvc(A_{pt}) \cup C_R$: Clocks to be reset are bound clock variables.
- $fvc(A_{pt}^1 | A_{pt}^2) = fvc(A_{pt}^1) \cup fvc(A_{pt}^2); \text{bvc}(A_{pt}^1 | A_{pt}^2) = \text{bvc}(A_{pt}^1) \cup \text{bvc}(A_{pt}^2)$.
- $fvc(A_{pt}^1 \oplus p A_{pt}^2) = fvc(A_{pt}^1) \cup fvc(A_{pt}^2); \text{bvc}(A_{pt}^1 \oplus p A_{pt}^2) = \text{bvc}(A_{pt}^1) \cup \text{bvc}(A_{pt}^2)$.

The free and bound clock variables of choices and parallel composition are the union of the free and bound clock variables of each process. The reason that the set of clock variables is divided into bound and free parts is to avoid conflicts of clock valuations. For instance, let us consider the process $x_c \leq 8 \triangleright (||x_c|| A_{pt})$, in which the clock $x_c$ is reset, and this affects the invariant $x_c \leq 8$. Further, in the parallel composition $(||x_c|| A_{pt}) \| (x_c \leq 8 \triangleright A'_{pt})$ the clock variable $x_c$ is the shared variable of the two processes, however, the reset of $x_c$ affects the behavior of process $(x_c \leq 8 \triangleright A'_{pt})$. This is undesirable since the operational semantics of a process also depends on the behavior of the environment (which is hard to control).

Hence, I define the notion of processes with non-conflict of clock variables, using the following inductive definition and the predicate $nvc$:

1. $nvc(A)$; 2. $nvc(X_{pt})$; 3. $nvc(\alpha^* \triangleright_{\pi} A_{pt})$ iff $nvc(A_{pt})$; 4. $nvc([C_R][A_{pt}])$ iff $nvc(A_{pt})$;
5. $nvc(\phi \hookrightarrow A_{pt})$; 6. $nvc(\phi \triangleright A_{pt})$: in both cases, if $nvc(A_{pt}) \land (\text{clock}(\phi) \cap \kappa(A_{pt}) = \emptyset)$

Rule 1 holds because an extended process $A$ does not include any clock variable. Rule 2 says that the recursive process invocation of plain processes is non-conflict because a plain process does not contain clock variables. Rule 3 comes from the fact that action $\alpha^*$ is free from clock variables. Rule 4 says that if clock resettings are placed outside (outermost) all invariants and guard constructs then they do not cause conflict. Rules 5 and 6 say that if guard and invariant constructs are placed outside then their clock variables cannot be reset within $A_{pt}$ to avoid conflict. For the full list of $nvc$ rules please my report [Th12, 2013].

In the following, for each $\text{crypto}^{\text{prob}}$ process I add rules that associate each process to the invariant and resetting functions $\vartheta$ and $\kappa$, respectively. Note that I only give the two most important rules in this dissertation, the full list can be found in [Th12, 2013].

\[
\text{rk. } \kappa([C_R] A_{pt}) = C_R \cup \kappa(A_{pt}) \\
\text{ri. } \vartheta(\phi \triangleright A_{pt}) = \vartheta(A_{pt}) \land \phi.
\]
Rule $r_k$ says that the set of clocks to be reset in $\kappa(\|C_R\| A_{pt})$ is $C_R$ and the clock resets occur in $A_{pt}$; and rule $r_i$ says that the invariant of process $\phi \triangleright A_{pt}$ is the intersection of $\phi$ and the invariant predicate in $A_{pt}$.

### 3.5.2 Operational Semantics

The formal semantics of $\text{crypt}_{\text{time}}^{\text{prob}}$ follows the semantics of probabilistic timed automata. Namely, a state $s$ is defined by the pair $(A_{pt}, v)$, where $v$ is the clock valuation at the location with label $A_{pt}$ with the time issues defined at the location. The initial state $s_0$ consists of the initial process and initial clock valuation, $(A_{pt}^0, v_0)$. Note that the initial process $A_{pt}^0$ is the initial status of a system behavior, while $v_0$ typically contains the clocks in the reset state. The operational semantics of $\text{crypt}_{\text{time}}^{\text{prob}}$ is defined by a probabilistic timed transition system (PTTS).

A probabilistic timed transition system can be seen as the labeled transition system extended with time and probabilistic constructs. In my model, I follow the concept of [24], [44], but I also improve them with language elements and a new definition of bisimilarity for proving/refuting the security properties of protocols.

**Definition 13.** Let $\sum$ be the set of actions. A probabilistic timed transition system is defined as the tuple $PTTS = (\mathcal{S}, \sum \times \mathbb{R}_{\geq 0} \times \Pi, s_0, \rightarrow_{PTTS}, \mathcal{U}, F)$ where

- $\mathcal{S}$ is a set of states, and $s_0$ is an initial state.
- $\rightarrow_{PTTS} \subseteq \mathcal{S} \times (\sum \times \mathbb{R}_{\geq 0} \times \Pi) \times \mathcal{S}$ is the set of probabilistic timed labeled transitions. A transition is defined between the source and target state, and the label of the transition is composed of the actions, the time stamp (duration), and the probability of the action. When $(\alpha^*, d, \pi) \in \sum \times \mathbb{R}_{\geq 0} \times \Pi$ I denote the transition from $s$ to $s'$ by $s \overset{\alpha^*(d), \pi}{\rightarrow}_{PTTS} s'$. The appearance of $\pi$ on the arrow means that the transition is performed with the probability according to the distribution $\pi$. The label $\alpha^*(d)$ says that performing either a visible $\alpha$ or invisible (silent) $\tau$ action ($\alpha^* = \alpha \cup \tau$) consumes $d$ time units. I interpret $d$ as the time for executing action $\alpha^*$, and there is no idling time at $s$ before performing an action.

- $\mathcal{U} \subseteq \mathbb{R}_{\geq 0} \times \mathcal{S}$ is the until predicate, and is defined at a state $s$ with a time duration $d$. Whenever $(d, s) \in \mathcal{U}$ I use the notation $\mathcal{U}_d$.

- The scheduler $F$ chooses non-deterministically the distribution of action transition steps.

The probabilistic timed transition system $PTTS$ should satisfy the two axioms Until and Delay (in both cases $\implies$ denotes logical implication):

**Until** \[ \forall d, d' \in \mathbb{R}_{\geq 0}, \mathcal{U}_d(s) \land (d' < d) \implies \mathcal{U}_{d'}(s) \]

**Delay** \[ \forall d \in \mathbb{R}_{\geq 0}, s \overset{\alpha^*(d), \pi}{\rightarrow}_{PTTS} s' \text{ for some } s' \implies \mathcal{U}_d(s) \]

These two axioms define formally the meaning of the notion delay and until. Basically, axiom Until says that if the system stays in state $s$ until $d$ time units then it also stays in this state before $d$. While the axiom Delay says that if the system performs an action $\alpha$ at time $d$ then it must wait until $d$. Note that the meaning of until differs from time invariant, because in case of until, the system waits (stay idled) at least $d$ time units in a state (location, if talking about automata), whilst invariant says that the system must leave the state (location) upon $d$ time units have elapsed (if it cannot move from the state then we get deadlock). In addition, $\mathcal{U}$ are the smallest set satisfying the following rules.
3. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WSN TRANSPORT PROTOCOLS

u1. $U_d(A, v)$; u2. $U_d(\alpha^* \prec \pi A_{pt}, v)$;

u3. $U_d(\phi \leftrightarrow A_{pt}, v)$ if $U_d(A_{pt}, v)$;

u4. $U_d([C_R\parallel A_{pt}, v)$ if $U_d(A_{pt}, v[rst : C_R])$;

u5. $U_d(\phi \triangleright A_{pt}, v)$ if $U_d(A_{pt}, v) \land \models (v + d)(\phi)$;

u6. $U_d(A_{pt}^1 [ ] A_{pt}^2, v)$ if $U_d(A_{pt}^1, v) \lor U_d(A_{pt}^2, v)$;

u7. $U_d(A_{pt}^1 \mid A_{pt}^2, v)$ if $U_d(A_{pt}^1, v) \lor U_d(A_{pt}^2, v)$;

u8. $U_d(A_{pt}^1 \oplus_p A_{pt}^2, v)$ if $U_d(A_{pt}^1, v) \lor U_d(A_{pt}^2, v)$;

u9. $U_d(X_{pt}, v)$ if $U_d(P[P/X_{pt}], v)$;

Rules (u1-u2) are the Until axioms for the states $(A, v)$ and $(\alpha^* \prec \pi A_{pt}, v)$. In u2 the system stays in the state $(\phi \leftrightarrow A_{pt}, v)$ until $d$ time units, if this is valid to the state $(A_{pt}, v)$ as well. Rules (u4-u5) come from the definition of the clock reset and invariant. In rule (u4) $v[rst : C_R]$ represents the clock valuation $v$ where the clocks in $C_R$ are reset. Rules (u6-u8) say that the system stays until $d$ time units at the state with $A_{pt}^1 [ ] A_{pt}^2, A_{pt}^1 \mid A_{pt}^2, A_{pt}^1 \oplus_p A_{pt}^2$, if it stays $d$ time in the state with one of the two processes $A_{pt}^1$ and $A_{pt}^2$. Rule u9 is concerned with the until predicate for (recursive) process variable $X_{pt}$, which comes directly from the definition of recursive process invocation. Note that $P$ is a plain process defined in crypt.

I define the satisfaction predicate $\models, \models \subseteq \Phi(C)$, on clock constraints. For each $\phi \in \Phi(C)$ I use the shorthand $\models v(\phi)$ iff $v$ satisfies $\phi$, for all clock valuation $v$. The set of past closed constraint, $\Phi(C) \subseteq \Phi(C)$, is used for defining semantics of location invariant, $\forall v \in V, d \in \mathbb{R}_{\geq 0}: \models (v + d)(\phi) \implies \models v(\phi)$. Intuitively, this says that if the valuation $v + d$, which is defined as $v(x_c) + d$ for all clocks $x_c$, satisfies the constraint $\phi$ then so does $v$. I adopt the variant of time automata used in [24], where location invariant and clock resets are defined as functions $\partial$ and $\kappa$ assigning a set of clock constraints $\Phi(C)$ and a set of clocks to be reset $R(C)$, respectively, to a $\text{crypt}^\text{prob}_\text{time}$ Process.

The probabilistic timed transition (action) rules for $\text{crypt}^\text{prob}_\text{time}$ are given as follows. I provide the connection of each PTTS transition with the edge syntax in probabilistic timed automata.

\begin{align*}
\text{a1. } (\alpha^* \prec \pi A_{pt}, v) & \xrightarrow{\alpha^*(d), \pi} \text{PTTS} \left( A_{pt}, v + d \right) \text{ if } \alpha^* \prec \pi A_{pt} \\
\text{a2. } ([C_R\parallel A_{pt}, v) & \xrightarrow{\alpha^*(d), \pi} \text{PTTS} \left( A'_{pt}, v' \right) \text{ if } (A_{pt}, v[rst : C_R]) \\
\text{a3. } (\phi \leftrightarrow A_{pt}, v) & \xrightarrow{\alpha^*(d), \pi} \text{PTTS} \left( A'_{pt}, v' \right) \text{ if } (A_{pt}, v) \\
\text{a4. } (\phi \triangleright A_{pt}, v) & \xrightarrow{\alpha^*(d), \pi} \text{PTTS} \left( A'_{pt}, v' \right) \text{ if } (A_{pt}, v) \\
\text{a5. } (A_{pt}, v) & \xrightarrow{\alpha^*(d), \pi} \text{PTTS} \left( \phi \triangleright A'_{pt}, v' \right) \text{ if } (A_{pt}, v) \\
\text{a6. } (A_{pt}^1 [ ] A_{pt}^2, v) & \xrightarrow{\alpha^*(d), \pi} \text{PTTS} \left( A'_{pt}, v' \right) \text{ if } (A_{pt}^1, v) \\
\text{a7/a. } (A_{pt}^1 \oplus_p A_{pt}^2, v) & \xrightarrow{\alpha^*(d), \pi(p)} \text{PTTS} \left( A'_{pt}, v' \right) \text{ if } A_{pt}^1 \oplus_p A_{pt}^2 \\
\end{align*}
3.5. \textit{crypt}^\text{prob\_time}: Extending \textit{crypt} with timed and probabilistic syntax and semantics

\begin{equation}
a1/a2. \quad (A_1^{pt} \oplus_p A_2^{pt}, v) \xrightarrow{\alpha^*(d), \pi}^{PTTS} (A_2^{pt}, v') \quad \text{if} \quad A_1^{pt} \oplus_p A_2^{pt} \xrightarrow{\alpha^*, \text{true}}^{\pi (1-p)} A_2^{pt}
\end{equation}

\begin{equation}
a3. \quad (A_1^{pt} | A_2^{pt}, v) \xrightarrow{\alpha^*(d), \pi}^{PTTS} (A_2^{pt} | \text{norst}(A_2^{pt}), v') \quad \text{if} \quad A_1^{pt} \xrightarrow{\alpha^*, \text{true}}^{\pi} (A_2^{pt}, v');
\end{equation}

\begin{equation}
a4. \quad (X_{pt}, v) \xrightarrow{\alpha^*(d), \pi}^{PTTS} (P', v') \quad \text{if} \quad (P|P/X_{pt}, v) \xrightarrow{\alpha^*, \text{true}}^{\pi (P', v')}.
\end{equation}

In rule a2, \( v' = v[rst : C_R] + d \), and in the rest rules, \( v' = v + d \). \( v[rst : C_R] \) represents the valuation \( v \) where the clocks in \( C_R \) are reset. Each rule should be interpreted that the PTTS transition on the left side can be performed if there is an edge in a corresponding automaton. For instance, rule a1 applies if there is an edge \( \alpha^* \prec_{\pi} A_{pt} \xrightarrow{\alpha^*, \text{true}}^{\pi} A_{pt} \) in the corresponding automaton. Rule a1 says that after performing action \( \alpha^* \) with \( d \) time units the system gets to the process \( A_{pt} \) with the clock valuation after \( d \) time units elapsed. Rule a2 says that by the time \( \|C_R\| A_{pt} \) proceeds to \( A_{pt} \), the clocks in \( C_R \) will have been reset. In the rules a3 and a4 the timed transition can be performed if \( (v + d)(\phi) \) holds, which means that the valuation \( v + d \) must satisfy the clock guard \( \phi \). Rules a5-a6 describe the case when process \( A^0_{pt} \) is activated (the rules for activating \( A_{pt} \) are similar). \( \pi(p) \) and \( \pi(1-p) \) in rules a7/a-b mean that in distribution \( \pi \) the first and second transitions (edges) are chosen with probability \( p \) and \( (1-p) \). In a5 to avoid conflict of clock variables, it is required that after performing the transition, process \( A^0_{pt} \) cannot perform resetting at the beginning. The last rule is the action rule for recursive process variable \( X_{pt} \). It can be proven, based on the rules a1-a9 and a1-a9, that probabilistic timed transition system of \( \text{crypt}^\text{prob\_time} \) satisfies axioms \textit{Until} and \textit{Delay}, hence, it is well defined.

\textbf{Theorem 2.} For all \( \text{crypt}^\text{prob\_time} \) process \( A_{pt} \) and for all closed valuation \( v_0, \) \( \text{PTTS}(A_{pt}, v_0, F) \) is indeed the probabilistic timed transition system defined in probabilistic timed automata. Hence, the correctness of the semantics of \( \text{crypt}^\text{prob\_time} \) is based on the correctness of the probabilistic timed automata.

\textbf{Proof.} (Sketch) Any process defined in \( \text{crypt\_time} \) can be expressed in a corresponding timed automaton. To show this, I adopt the notion \textit{image-finite} and \textit{finiteley sorted} (borrowed from transition system theory). A probabilistic timed automaton is image-finite if for a given distribution \( \pi \), the set of outgoing edges of each state with the same action \( act \) is finite. Formally, for each \( q, act \) and \( \pi \), the size of the set \( \{ q \xrightarrow{\text{act}, \phi} q' \ | \ q \in \mathcal{L} \} \) is finite. A probabilistic timed automaton is finiteley-sorted if for a given distribution \( \pi \), the set of outgoing edges with the same action \( act \) of every state, \( \{ act \ | \ \exists q' \in \mathcal{L} : q \xrightarrow{\text{act}, \phi} q' \} \), is finite.

The associated probabilistic timed automaton for a (initial) process \( A^0_{pt} \) can be constructed by associating the process \( A^0_{pt} \) to the initial location \( q_0 \), then each transition \( A^0_{pt} \xrightarrow{\alpha^*, \phi} A^1_{pt} \) can be defined in terms of a corresponding probabilistic timed automaton, \( PT = (\mathcal{L}, q_0, \sum, C, \partial, \kappa, \tau, E, II) \), as follows:

\begin{equation}
A^0_{pt} = \|\kappa(q_0)\| \partial(q_0) \triangleright (\phi \leftrightarrow (\alpha^* \prec_{\pi} A^1_{pt}))
\end{equation}

In this process definition, \( A^0_{pt} \) corresponds to location \( q_0 \) of the corresponding probabilistic timed automaton at which the set of clocks to be reset is \( \kappa(q_0) \), and the invariant \( \partial(q_0) \) is defined. The edge from \( q_0 \) to \( q_1 \), \( q_0 \xrightarrow{\alpha^*, \phi, \pi} q_1 \), corresponds to the time construct \( \phi \leftrightarrow (\alpha^* \prec_{\pi} A^1_{pt}) \). Generally, for every subsequent process \( A^i_{pt} \) after some transition steps from \( A^0_{pt} \) we have

\begin{equation}
A^i_{pt} = \|\kappa(q_i)\| \partial(q_i) \triangleright (\phi \leftrightarrow (\alpha^* \prec_{\pi} A^{i+1}_{pt}))
\end{equation}

which corresponds to the edge \( q_i \xrightarrow{\alpha^*, \phi, \pi} q_{i+1} \) in \( PT \). For the more complex target process such as \( A_{pt}^{(i+1)} | \[ \ldots[ ] \ A_{pt}^{(i+1)n} \) we have

83
where $A^i_{pt}$ corresponds to location $q_i$ (with the appropriate resets and invariant) and the sub-process $[\prod_{j=1}^{n} (\phi_j \rightarrow (\alpha_j^* \succeq_{\pi} A_{pt}^{i+1}))]$ corresponds to the edge from $q_i$ to the location $q_{i+1}$ with label $(\alpha_i^*, \phi_j)$, $1 \leq j \leq n$, such that $\alpha_j^*$ is the first enabled action (due to the valid condition at $q_j$) among the $n$ processes. In case there are more than one enabled action at the same time, it can be treated in the same way as the non-deterministic choices.

For the target process $A_{pt}^{i+1}$, we have the following process definition for $A^i_{pt}$:

$$A^i_{pt} = \|\kappa(q_i)\| \cdot (\partial(q_i) \triangleright (\phi_1 \Leftarrow (\alpha_1^* \succeq_{\pi} A_{pt}^{i+1}))) \oplus_p A^i_{pt}$$

where $A^i_{pt}$ corresponds to location $q_i$, and from this location we can get to the location $A_{pt}^{i+1}$ with probability $p$ and to $A_{pt}^{i+1}$ with probability $1-p$, via the edges (or the corresponding transition) with the labels $(\alpha_1^*, \phi_1)$ and $(\alpha_2^*, \phi_2)$, respectively.

For the target process $A_{pt}^{i+1}$, we have

$$A^i_{pt} = \|\kappa(q_i)\| \cdot (\partial(q_i) \triangleright (\phi \Leftarrow (\alpha^* \succeq_{\pi} A_{pt}^{i+1})))$$

which says that from the location corresponding to process $A^i_{pt}$ we can get to the location corresponding to $A^{i+1}_{pt} | \cdots | A^{i+1}_n$, via the edge (transition) with the label $(\alpha^*, \phi)$. The reason is that we associate the parallel composition of crypt\textsubscript{prob\_time}, processes to a location in the probabilistic timed automata, instead of interpreting it as parallel composition of automata. Hence, the transition $A^i_{pt} | A^2_{pt} \xrightarrow{\alpha^*, \phi} A^{i+1}_{pt} | A^2_{pt}$ corresponds to the edge $q \xrightarrow{\alpha^*, \phi} q'$, where $A^i_{pt} | A^2_{pt}$ corresponds to location $q_i$, while $A^{i+1}_{pt} | A^2_{pt}$ to $q'$. In case there is not any outgoing edge from $q_i$ we have the following process definitions for each type of target process:

$$A^i_{pt} = \|\kappa(q_i)\| \cdot \partial(q_i) \triangleright \textbf{nil}$$

I defined rules for renaming of clock variables and I showed that the process with non-conflict of clock variables, ncv, property is preserved by clock renaming, hence, the restriction I made to process without conflict of clock variables is harmless [24]. Based on the rules of renaming I also added new rules for structural equivalent resulted from renaming. I omit the discussion of these rules in details because I do not use them in the dissertation, but the reader can find it in my longer report [Th12, 2013].

**Weak probabilistic timed (weak prob-timed) labeled bisimulation**

I provide a novel bisimilarity definition, called weak prob-timed labeled bisimulation for crypt\textsubscript{prob\_time}, which enables us to prove or refute the security of probabilistic timed systems.

My proposed definition makes use of the definition of static equivalence proposed in the applied \(\pi\)-calculus [27], which says that the outputs of static equivalent processes cannot be distinguished by the environment (or attackers). The main advantage of static equivalence is that it only takes into account the static knowledge exposed by two processes to show the behavioral equivalence of them. This method is much easier to use than using the observational equivalence [27], where I have to consider the dynamic behavior of processes.

Let the extended process $A$ be $\{x_1/x_1\} \mid \cdots \mid \{x_n/x_n\} \mid P_1 \mid \cdots \mid P_n$. The frame $\varphi$ of $A$ is the parallel composition $\{x_1/x_1\} \mid \cdots \mid \{x_n/x_n\}$ that models all the information that is output so far by the process $A$, which are $t_1, \ldots, t_n$ in this case.
Definition 14. **Static equivalence for extended processes** ($\approx_s$). Two extended processes $A^1$ and $A^2$ are statically equivalent, denoted as $A^1 \approx_s A^2$, if their frames are statically equivalent. Two frames $\varphi_1$ and $\varphi_2$ are statically equivalent if they include the same number of active substitutions and same domain; and any two terms that are equal in $\varphi_1$ are equal in $\varphi_2$ as well. Intuitively, this means that the outputs of the two processes cannot be distinguished by the environment.

In my proposed weak prob-timed labeled bisimulation, I extend the static equivalence with time and probabilistic elements. The meaning of weak is that in this dissertation I want to examine whether the attackers can distinguish the behavior of two processes, based on the information they can observe. Hence, in weak prob-timed labeled bisimulation, I do not require the equivalence of the probability of two action traces, because practically an observer cannot distinguish if an action is performed with 1/2 or 1/3 probability.

Nevertheless, I also proposed the definition of strong prob-timed labeled bisimulation in my longer report [Thi12, 2013], which I do not discuss in this dissertation, because I found that for analyzing the security of DTSSN and SDTP, it is sufficient to use the weak prob-timed labeled bisimulation. **Strong prob-timed labeled bisimulation** is stricter, since it also distinguishes two processes based on the probability of their corresponding action traces.

Definition 15. **(Weak prob-timed labeled bisimulation for crypt$^{\text{prob}}$ processes)**

Let $\text{PTTS}(A^i_{pt}, v_0, F) = (S_i, \alpha \times \mathbb{R}^{\geq 0} \times \Pi, s_0^i, \rightsquigarrow_{\text{PTTS}}, \mathcal{U}^i, F)$, $i \in \{1, 2\}$ be two probabilistic timed transition systems for crypt$^{\text{prob}}$ processes. Weak prob-timed labeled bisimilarity ($\approx_{pt}$) is the largest symmetric relation $R$, $R \subseteq S_1 \times S_2$ with $s_0^1 R s_0^2$, where each $s^i$ is the pair of a closed crypt$^{\text{prob}}$ process and a same initial valuation $v_0 \in \mathcal{V}^i$, $(A^i_{pt}, v_0)$, such that $s_1 R s_2$ implies:

1. $A^1 \approx_s A^2$;
2. if $s_1 \xrightarrow{\tau(d), \pi} \text{PTTS}_1 s'_1$ for a scheduler $F$, then $\exists s'_2$ such that $s_2 \xrightarrow{\tau(\Sigma d_j), \pi}\text{PTTS}_2 s'_2$ for the same $F$, with $d = f(\Sigma d_j)$ for some function $f$, and $s'_1 R s'_2$;
3. if $s_1 \xrightarrow{\alpha(d), \pi} \text{PTTS}_1 s'_1$ for a scheduler $F$ and $fv(\alpha) \subseteq \text{dom}(A^1) \land bn(\alpha) \cap fn(A^2) = \emptyset$, then $\exists s'_2$ such that $s_2 \xrightarrow{\alpha(\Sigma d_j), \pi}\text{PTTS}_2 s'_2$ for the same $F$, with $d = f(\Sigma d_j)$ for some function $f$, and $s'_1 R s'_2$. Again, $\text{dom}(A^1)$ represents the domain of $A^1$, and vice versa. $A^1$ and $A^2$ are the extended processes we get by removing all the probabilistic and timed elements from $A^i_{pt}$ and $A^i_{pt}$, respectively.

The arrow $\xrightarrow{\tau,\pi}_{\text{PTTS}}$ is the shorthand of the action trace $\xrightarrow{\tau,\pi}_{\text{PTTS}} \xrightarrow{\alpha}_{\text{PTTS}} \xrightarrow{\tau,\pi}_{\text{PTTS}}$, where $\xrightarrow{\tau,\pi}_{\text{PTTS}}$ represents a series (formally, a transitive closure) of sequential transitions $\xrightarrow{\tau}_{\text{PTTS}}$. $\Sigma d_i$ on $\xrightarrow{\tau}_{\text{PTTS}}$ is the sum of the time elapsed at each transition, and represents the total time elapsed during the sequence of transitions. Note that $fn(A^i_{pt})$ and $\text{dom}(A^i_{pt})$ is the same as $fn(A^i)$ and $\text{dom}(A^1)$, respectively. Moreover, a process $A_{pt}$ is closed if its non-timed and “non-probabilistic” counterpart $A$ is closed.

Intuitively, in case $A^i_{pt}$ and $A^i_{pt}$ represent two protocols (or two variants of a protocol), then the meaning of each point in the Definition is as follows: (i) the outputs of the two processes $A^i_{pt}$ and $A^i_{pt}$ cannot be distinguished by the environment based on their behaviors; (ii) the time that the protocols spend on the performed operations until they reach the corresponding points is in some relationship defined by a function $f$. Here $f$ depends on the specific definition of the security property, for instance, it can return $d$ itself, hence, the requirement for time consumption would be $d = \Sigma d_i$. In particular, the first point means that $A^i_{pt}$ and $A^i_{pt}$ are statically equivalent, that is, the environment cannot distinguish the behavior of the two protocols based on their outputs; the second point says that $A^i_{pt}$ and $A^i_{pt}$ remain statically equivalent after silent transition (internal computation) steps. Finally, the third point says that the behavior of the two protocols matches in transition with the action $\alpha$. 


In this section, I demonstrate the application of crypt\textsuperscript{prob} for specifying the DTSN, the SDTP and the SDTP\textsuperscript{+} protocols. The main difference between the specification of WSN transport protocols and wireless ad-hoc networks routing protocols (in Section 2) is that in transport protocols the network topology for a given session is fixed. In addition, instead of broadcasting messages, the communication between two nodes takes place on the symmetric channel defined between the two nodes.

3.6.1 DTSN in crypt\textsuperscript{prob}

In order to shorten the presentation and making the dissertation more readable I only discuss the most important parts of the protocol. For the detailed descriptions, the reader is referred to the report [Th12 , 2013].

I assume the network topology $S-I-D$, where “−” represents a bi-directional link, while $S$, $I$, $D$ denote the source, an intermediate node, and the destination node, respectively. I also include the presence of the application that uses DTSN and SDTP, because it sends packet delivery requests to the source, and it receives delivered packets. In the rest of the dissertation I refer to the application as the upper layer. Note that the attack scenarios which can be found and proved in this topology is also valid in other topologies including more intermediate nodes. Moreover, I assume that each node has three cache entries, denoted by $e^a$, $e^b$, and $e^c$, $1 \leq k \leq 3$. For brevity let $e^a_{1-3}$ range over $e^a$ from index 1 to 3, and the same is true for $e^b_{1-3}$ and $e^c_{1-3}$. I define symmetric public channels between the upper layer and the source, $e_{sup}$; the upper layer and the destination, $e_{dup}$; the source and the intermediate node, $e_{si}$; the intermediate node and the destination $e_{id}$. The public channels $e_{siACK}$, $e_{siNACK}$, $e_{idACK}$ and $e_{idNACK}$ are defined for sending and receiving ACK and NACK messages between the source and the intermediate, and between the destination and the intermediate nodes, respectively. Moreover, I define additional channels $e_{error}$ and $e_{sessionEND}$ for sending and receiving error and session-end signals.

I define crypt\textsuperscript{prob} processes upLayer, Src, Int, Dst for specifying the behavior of the upper layer, the source, intermediate, and destination nodes. The DTSN protocol for the given topology is specified by the parallel composition of these four processes.

The specification of the DTSN protocol:

\[ \text{Prot}(\text{params}) \overset{\text{def}}{=} \begin{cases} & \text{let } (e^a_1, e^2_1, e^3_1, e^a_2, e^2_2, e^3_2, e^a_3, e^2_3, e^3_3, \text{cntsq}) = (\otimes, \otimes, \otimes, \otimes, \otimes, \otimes, \otimes, \otimes, \otimes, 1) \\ & \text{in INITDTSN}(); \\ & \text{INITDTSN}() \overset{\text{def}}{=} e_{sup}(\text{cntsq}), \text{DTSN}(\text{params}) \\ & \text{DTSN}(\text{params}) \overset{\text{def}}{=} \\ & \quad \text{upLayer}(\text{incr}(\text{cntsq})) | \text{initSrc}(s, d, \text{apID}, e^a_{1-3}, sID, \text{earAtmp}) | \\ & \quad \text{Int}(e^a_{1-3}) | \text{Dst}(e^a_{1-3}, \text{ackNbr}, \text{nackNbr}, \text{toRTX1}, \text{nxtsq}) \end{cases} \]

I refer to the tuple of parameters $(\text{cntsq}, s, d, \text{apID}, e^a_{1-3}, sID, \text{earAtmp}, e^a_{1-3}, e^c_{1-3}, \text{ackNbr}, \text{nackNbr}, \text{toRTX1}, \text{nxtsq})$ by $\text{(params)}$. The process $\text{Prot}(\text{params})$ describes DTSN with variable initializations. The let construct is used to initialize the value of the cache entries to $\otimes$, and the current sequence number to 1. The unique name $\otimes$ is used to represent the empty content. In the following, I give a brief overview of the main processes of the formal specifications. Each main process is composed of additional sub-processes, which I skip discussing here. The processes are recursively invoked to model process replication. Interested readers can find the full description in [Th12 , 2013].

I introduce two clock variables: $a_a$ for the activity timer, and $a_c$ for the EAR timer. According to the specification of the DTSN protocol [47], to model timeout I make use of the clock
invariant defined on the process $Src$. The initial state of DTSN for the given topology is specified as the process $\begin{cases} x^c \leq T^{act} \Rightarrow \text{initFwdDt}(s, d, \text{apID}, e_{1-3}^s, \text{ID}, x^c) \\ \{ x^c \leq T^{act} \Rightarrow \text{initRcvACKS}(s, d, \text{apID}, e_{1-3}^s, \text{ID}, \text{earAtmp}) \\ x^c \leq T^{act} \Rightarrow \text{initRcvACKS}(s, d, \text{apID}, e_{1-3}^s, \text{ID}, \text{earAtmp}) \end{cases}$

In process $\text{INITDTSN}()$, first of all, the request for sending the first packet with sequence number $cntsq$ is sent. Then, the next request, $cntsq + 1$, is expected to be sent by the upper layer, which is specified by the process $\text{upLayer}(\text{incr}(cntsq))$. The parameters of process $Src$ are composed of the IDs of the source and the destination; the application ID; the three cache entries, the session ID; and the latest number of $EAR$ attempts. Process $Init$ has the content of the three cache entries as parameter. The parameters of process $Dst$ include the cache entries; the $ACK/NACK$ numbers for composing acknowledgement messages; the packet to be re-transmitted and the next expected packet.

The source handling the activity timer expiration:

\begin{align*}
\text{InitSrc}(s, d, \text{apID}, e_{1-3}^s, \text{ID}, \text{earAtmp}) & \overset{def}{=} \\
1. c^{sup}(x^c) & = \begin{cases} x^c < T^{act} \Rightarrow \text{initFwdDt}(s, d, \text{apID}, e_{1-3}^s, \text{ID}, x^c) \\ x^c \leq T^{act} \Rightarrow \text{initRcvACKS}(s, d, \text{apID}, e_{1-3}^s, \text{ID}, \text{earAtmp}) \\ x^c \leq T^{act} \Rightarrow \text{initRcvACKS}(s, d, \text{apID}, e_{1-3}^s, \text{ID}, \text{earAtmp}) \end{cases} \\
2. & [x^c < T^{act} \Rightarrow (x^c = T^{act}) \Rightarrow \text{cSessionEND}(\text{SEND}) \cdot \text{nil}] \\
3. & [x^c \leq T^{act} \Rightarrow (x^c = T^{act}) \Rightarrow \text{actTimeOut}] 
\end{align*}

The specification of the source node: The process $\text{initSrc}$ specifies the behavior of the source node when it starts to handle the first packet. In case of the first packet within a session, the source has not launched the EAR timer yet, but only the $ACT$ (activity) timer. The launching of the $ACT$ timer is defined by the clock invariant construct, $\{ x^c < T^{act} \Rightarrow \text{initFwdDt}(s, d, \text{apID}, e_{1-3}^s, \text{ID}, x^c) \}$. The part $(x^c < T^{act}) \Rightarrow \text{initRcvACKS}(s, d, \text{apID}, e_{1-3}^s, \text{ID}, \text{earAtmp})$ represents the guarded action with the time guard $(x^c < T^{act})$. The sub-process $\text{actTimeOut}$ describes the behavior of the protocol after the $ACT$ timer has expired. Similarly, for the EAR timer I define the process $\text{earTimeOut}$.

The three choice options represent the “wait for event” activity of the source. The choice is resolved after the corresponding event occurs. Each choice option represents a scenario: The last (third) option in point 3 describes the case when the activity timer has elapsed. The process $\{ x^c < T^{act} \Rightarrow (x^c = T^{act}) \Rightarrow \text{actTimeOut} \}$ follows the concept of the timed automaton, and says that when the $ACT$ timer has elapsed the protocol proceeds with the process $\text{actTimeOut}$, which describes the defined behavior of $Src$ after timeout. The second choice (in point 2.) is for the case when the session is terminated, which happens when the constant $SEND$ has been sent on the private channel $\text{cSessionEnd}$ by the source [47]. Each node is defined such that it waits for $SEND$ by the construct $\text{cSessionEND}(\text{SEND})$, where $\text{SEND}$ is used to check if the received data is equal to $SEND$. This construct is basically the abbreviation of the process $\text{cSessionEND}(\text{xSend})$. $\{ x^{\text{SEND}} = \text{SEND} \}$. After receiving the session end signal each node terminates its operation.

The reason of setting $\text{cSessionEnd}$ to be a private channel, and making it not accessible for the attackers, is because I do not allow the attackers to send the session end signal directly to honest nodes. I assume that the session termination cannot be interrupted by the timeouts, basically, it can be seen as an atomic action. When the first option (in point 1.) has been chosen, it means that the source received the delivery request for the packet with the sequence number $x^c$ from the upper layer, and this input action is not interrupted by a session end or a timeout.

87
The similar choice options among the possible actions (like in points 1-3) are repeatedly put into the processes `initFwdDt`, `initRcvACKS`, and `initRcvNACKS`. These three processes specify how the source node forwards the first data packet, how the source handles the first received ACK and NACK packet, respectively. The choice options among the processes for handling the session end, the activity, and/or the EAR timeouts are placed before the action steps to be performed at that point. Specifically, in `initFwdDt(s, d, apID, e^f_{i-3}, sID, sq)`, after checking that the buffer is not full, the source stores the packet in a cache entry with let `e^f_{i} = (s, d, apID, sID, sq)`, then, after resetting the ACT timer by the construct `∥ x^{act}_{e} ⊥`, it proceeds to checking the fullness of the buffer in the parametric process `checkBuffandAW`. In point 4, the received packet is stored in one of the cache entries, `i ∈ \{1, 2, 3\}`. The process `∥ x^{act}_{e} ⊥ checkBuffandAW` resets the ACT timer and continues with checking whether the buffer is full and if sq is AW multiple.

**The source stores the received packet and resets the ACT timer:**

\[
\text{initFwdDt}(s, d, \text{apID}, e^f_{i-3}, \text{sID}, \text{sq}) \overset{def}{=} \\
\{ x^{act}_{e} ≤ T^{act} \} \triangleright (x^{act}_{e} ≤ T^{act}) \iff \text{let } e^f_{i} = (s, d, \text{apID}, \text{sID}, \text{sq}) \text{ in} \\
\{ x^{act}_{e} ≤ T^{act} \} \triangleright (x^{act}_{e} ≤ T^{act}) \iff \text{checkBuffandAW}(s, d, \text{apID}, e^f_{i-3}, \text{sID}, \text{sq}, \text{earAtmp})
\]

Process `Src(s, d, apID, e^f_{i-3}, sID, earAtmp)` is a bit differ from `initSrc(s, d, apID, e^f_{i-3}, sID, earAtmp)`, in that the EAR timer will be launched in it. `Src` considers the case when the source has already stored some packets, and is located within the process `checkBuffandAW` after the source sent the current data packet, and it is located in `initRcvACKS`, and `initRcvNACKS` after the source has finished deleting its buffer and re-transmitting the required packets, according to the received ACK and NACK packets.

**The source’s activity after it has already stored packets :**

\[
\text{Src}(s, d, \text{apID}, e^f_{i-3}, \text{sID}, \text{earAtmp}) \overset{def}{=} \\
\{ x^{act}_{e} ≤ T^{act}, x^{ear}_{e} ≤ T^{ear} \} \triangleright (x^{act}_{e} ≤ T^{act}, x^{ear}_{e} ≤ T^{ear}) \iff \text{fwdDt}(s, d, \text{apID}, e^{f}_{i-3}, \text{sID}, \text{sq})
\]

\[
∥ x^{act}_{e} ≤ T^{act}, x^{ear}_{e} ≤ T^{ear} \} \triangleright (x^{act}_{e} ≤ T^{act}, x^{ear}_{e} ≤ T^{ear}) \iff \text{rcvACKS}(s, d, \text{apID}, e^{f}_{i-3}, \text{sID}, \text{earAtmp})
\]

\[  \{ x^{act}_{e} ≤ T^{act}, x^{ear}_{e} ≤ T^{ear} \} \triangleright (x^{act}_{e} ≤ T^{act}, x^{ear}_{e} ≤ T^{ear}) \iff \text{rcvNACKS}(s, d, \text{apID}, e^{f}_{i-3}, \text{sID}, \text{earAtmp}) \]

Points 5-8 also include the reseting of the EAR timer, hence, the clock invariant `{ x^{act}_{e} ≤ T^{act} }` in points 1-3 is extended with `{ x^{ear}_{e} ≤ T^{ear} }`, and the action guard becomes `{ x^{act}_{e} ≤ T^{act}, x^{ear}_{e} ≤ T^{ear} }`. In addition, the processes `fwdDt`, `rcvACKS` and `rcvNACKS` differ from `initFwdDt`, `initRcvACKS`, and `initRcvNACKS`, such that in the first three processes, the source has to perform some searching steps. Finally, in process `Src`, the timeout of the EAR timer should be taken into account (in point 8). The process `Src` is located at the end of the processes `fwdDt`, `rcvACKS` and `rcvNACKS` to model the recursive behavior of the source, until the session end.

In the following, I define the processes `actTimeOut` and `earTimeOut` that describe the defined behavior of the source when the ACT and the EAR timers have elapses, respectively.
3.6. Specifying WSN transport protocols in $\text{crypt}_\text{prob}$

The source handles activity timer expiration:

\[
\text{actTimeOut} \triangleq
\begin{cases}
\text{nbrUnConfirmed} = 0 & \overrightarrow{\text{sessionEND}} (\text{SEND}).\text{nil} \\
\text{nbrUnConfirmed} > 0 & \overrightarrow{\text{EAR}}.
\end{cases}
\]

\[
\| x_{\text{act}}, x_{\text{ear}} \| \{ x_{\text{act}} \leq T_{\text{act}}, x_{\text{ear}} \leq T_{\text{ear}} \} \rightarrow \text{Src}(s, d, \text{apID}, e_{1-3}, \text{sID}, \text{earAtmp});
\]

In \text{actTimeOut}, according the definition of DTSN, if there is not any unconfirmed packet in the buffer, the session is terminated. Otherwise, the EAR packet is sent on channel $c_{\text{si}}$, then, the ACT and EAR timers are reset, followed by waiting for an event after invoking recursively the process $\text{Src}$.

the source handles EAR timer expiration:

\[
\text{earTimeOut} \triangleq
\begin{cases}
\text{let earAtmp = incr(earAtmp) in} \\
\{| \text{earAtmp} > \text{earMAX} \} \overrightarrow{\text{sessionEND}} (\text{SEND}).\text{nil} \\
\{| \text{earAtmp} \leq \text{earMAX} \} \overrightarrow{\text{EAR}}.
\end{cases}
\]

\[
\| x_{\text{act}}, x_{\text{ear}} \| \{ x_{\text{act}} \leq T_{\text{act}}, x_{\text{ear}} \leq T_{\text{ear}} \} \rightarrow \text{Src}(s, d, \text{apID}, e_{1-3}, \text{sID}, \text{earAtmp})
\]

In process \text{earTimeOut}, the EAR attempt is increased. Then, if the EAR attempt exceeds the MAX number, the session is terminates, otherwise, the ACT and EAR timers are reset, and it waits for events by invoking recursively the process $\text{Src}$.

The specification of the intermediate node: In process $\text{Int}$, the choices options have the following meaning: (i) the intermediate node may receive a data packet on the channel $c_{\text{si}}$, and after that it handles these received packet according to the definition of DTSN, or (ii) it can receive and handle an ACK or (iii) handling the received NACK message, and finally, (iv) it can terminate its operation when it gets the session end signal (i.e., the constant $\text{SEND}$).

I also add a probabilistic choice in the specification. According to the definition of the DTSN protocol, the probabilistic choice is placed within process $\text{Int}(e_{1-3})$, which is the specification of node $I$. In particular, after receiving a packet, an intermediate node stores the packet in its cache with probability $p$. To model this behavior, I add the probabilistic choice construct in the sub-process $\text{hndleDtI}$, which is responsible for handling a received data packet. Let us denote the tuple of process parameters, $(x_s, x_d, x_{\text{apID}}, x_{\text{sID}}, x_{\text{sq}}, x_{\text{ear}}, x_{\text{rtx}}, e_{1-3})$, be $\text{params}_{\text{strdt}}$.

\[
\text{strAndFwI}(\text{params}_{\text{strdt}}) \oplus_p \text{FwI}(\text{params}_{\text{strdt}});
\]

Process $\text{strAndFwI}$, which describes the case when the intermediate node stores (and forwards) the received packet, is chosen with probability $p$, and process $\text{FwI}$ that specifies the only-forwarding case, is selected with probability $1 - p$.

The specification of the destination node: For the process $\text{Dst}$, the destination can either receive a data packet on the channel $c_{\text{id}}$ or receive a session end signal. In the first case, $\text{Dst}$ proceeds to $\text{hndleDtDst}$, in which the destination performs the verification steps and delivers the packet to the upper layer, or sends an ACK or a NACK.
Process that models the behavior of the destination:

```
Dst(c_{t-3}^e, ackNbr, nackNbr, toRTX1, nxtsq) def/
c_{id}((x_s, x_d, x_apID, x_SID, x_sq, x_ear, x_rtx)).handleDtDst [] c_{sessionEND}(=SEND).nil;
```

### 3.6.2 SDTP in crypt\textsuperscript{prob}

The non-cryptographic parts of SDTP including the timing and probabilistic elements are specified in the same way as in case of the DTSN protocol. Hence, I focus on the cryptographic parts of SDTP. To model the cryptographic primitives and operations in SDTP, I add the following equations into the set of equational theories:

- **Functions:**
  
  \[ K(n, ACK); K(n, NACK); mac(t, K(n, ACK)); mac(t, K(n, NACK)); \]

- **Equations:**
  
  - `checkmac(mac(t, K(n, ACK)), K(n, ACK)) = ok;`
  - `checkmac(mac(t, K(n, NACK)), K(n, NACK)) = ok.`

where functions `K(n, ACK)` and `K(n, NACK)` specify the ACK and NACK per-packet keys corresponding to the packet with sequence number `n`. The functions `mac(t, K(n, ACK))` and `mac(t, K(n, NACK))` compute the MAC on the message `t`, using the ACK and the NACK keys for the `n`-th packet. In order to simplify the modeling procedure, without violating the correctness of SDTP, I make an abstraction of the key hierarchy given in [17], where the per-packet keys are computed with a one-way function based on the shared secret unknown to the attacker. Instead, I assume that `K(n, ACK)` and `K(n, NACK)` cannot be generated (but can be intercepted) by the attackers. The attackers can only generate keys that are different from these keys. With this, I model the fact that the shared secret will never be revealed during the protocol.

The specification of SDTP is defined with the process `ProtSDTP(params)`, where `params` is the same parameter list specified in DTSN. Below, I discuss the main differences of SDTP compared to DTSN. To model the SDTP protocol, I extend the specification of the DTSN protocol in the following way. First, the source node extends each packet with an ACK MAC and a NACK MAC, then sends it to node `I`, which is accomplished by the following code part in the processes `initSrc` and `Src`:

The source sends a data packet in SDTP

```
let (ear, rtx, earAtmp) = (val1, val2, val3) in 
let (K_{ack}^{seq}, K_{nack}^{seq}) = (K(sq, ACK), K(sq, NACK)) in 
let ACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), K_{ack}^{seq}) in 
let NACKMAC = MAC((s, d, apID, sID, sq, ear, rtx), K_{nack}^{seq}) in 
\text{handleSrc}((s, d, apID, sID, sq, ear, rtx, ACKMAC, NACKMAC)).
```

In the first row, the variables `ear`, `rtx`, and `earAtmp` are given some values `val1`, `val2`, and `val3`, respectively. In the second row the `ACK/NACK` keys `K_{ack}` and `K_{nack}` are generated, while in the third and fourth rows the `ACK/NACK` MACs are computed using the generated `ACK/NACK` keys.

In addition, the specification of DSTN is extended with the verification of ACK MACs and NACK MACs when the source receives ACK and NACK packets. Formally, I define the process `recvACKS` as follows:

```
recvACKS(s, d, apID, e_{1-3}^s, sID, earAtmp) def/
```
The expected data on channel $c_{siACK}$ is the tuple of the ack number along with the per-packet ack key $ackkey$, and the nack key $nackkey$. This tuple is also included as the parameters of process $hndleACK$ and its sub-processes, which are responsible for handling the received $ACK$. I define the specification of $hndleACK$ with the verification of the stored ACK MAC using the keys included in the received ACK packets. This is modeled by the if construct in $crypt_{time}$, namely, $[checkmac(6(e_i^s), ackkey) = ok]$. In particular, $checkmac(6(e_i^s), ackkey)$ is the verification of the received ACK MAC, which is stored in the 6-th place in the cache entry $e_i^s$. The function $6(e_i^s)$ extracts the 6-th element of the cache entry $e_i^s$. The same verification is applied in the sub-processes.

When a NACK packet has been received the SDTP protocol includes verification of ACK MAC and NACK MACs. The structure of the NACK packet compared to DTSN case is extended with an ACK key (if any) and some NACK keys depending on the number of bits in the NACK packet. Hence, the expected data on channel $c_{siNACK}$ is extended with the ackkey and nackkey parameters, for instance, instead of $c_{siNACK}(acknum, b1)$ we have $c_{siNACK}(acknum, b1, ackkey, nackkey1)$. Namely, the verification part $[5(e_i^s) \leq acknum]$, which examines if the 5-th element of the entry $e_i^s$ (i.e. the stored sequence number $sq$) is less or equal to the received $acknum$, is extended with the verification of the ACK/NACK MACs $[checkmac(6(e_i^s),ackkey) = ok]$ $[checkmac(7(e_i^s),nackkey) = ok]$ for each $i \in \{1, 2, 3\}$.

In addition, the $crypt_{time}$ processes $Src$ and $Int$ are specified such that whenever the MAC verification made by $S$ and $I$ on the received ACK/NACK message fails, they output the predefined constant $BadControl$ via the public channel $c_{badPKK}$. Finally, in $ProtSDTP(params)$, the destination process $Dst$ is defined such that it outputs the predefined constant $BadData$ on channel $c_{badPKK}$, whenever the verifications of the ACK and NACK MACs it performed on the received data packet fails.

### 3.6.3 SDTP$^+$ in $crypt_{time}$

Again, the non-cryptographic parts of SDTP$^+$ including the timing and probabilistic elements are specified in the same way as in the DTSN protocol. I focus on the processes that are related to the security mechanism of SDTP$^+$. SDTP$^+$ uses all the cryptographic primitives and operations defined in SDTP, and in addition to these, the following special names, functions and equations are also required for specifying SDTP$^+$:

- **Names**: $sk_{src}$, $pk_{src}$, $K_{ack}$, $K_{nack}$, $K_{sd}$;
- **Functions**: $sign(t, sk_{src})$;
- **Equations**: $checksign(sign(t, sk_{src}), pk_{src}) = ok$;

where $sk_{src}$ and $pk_{src}$ represent the secret and public key of the source node, $K_{ack}$, $K_{nack}$ and $K_{sd}$ represent the ACK/NACK master keys, and the shared key of the source and the destination for a given session, which are freshly generated at the beginning of each session. The functions $sign(t, sk_{src})$ and $H(t)$ define the digital signature computed on the message $t$ using the secret key $sk_{src}$, and the one-way hash computed on $t$, respectively. The equation $checksign(sign(t, sk_{src}), pk_{src}) = ok$ defines the signature verification, using the corresponding public key $pk_{src}$. I do not define an equation for the hash function $H(t)$ in order to ensure its one-way property. Namely, $H(t)$ does not have a corresponding inverse function which returns $t$, and $H(t_1) = H(t_2)$ holds only when $t_1$ and $t_2$ are the same.

The specification of the SDTP$^+$ protocol is defined by the process $ProtSDTPplus(params)$, where $params$ is the same parameter list specified in DTSN and SDTP. In the following, I only discuss the main differences of SDTP$^+$ compared to DTSN and SDTP. According to the specification of SDTP$^+$, I examine the activities of each node, which are related to the hash-chain and
Merkle-tree. I assume that a session contains four packets with the sequence number from 1 to 4. The source node, besides computing the ACK and NACK master keys $K_{\text{ACK}}$ and $K_{\text{NACK}}$, it also generates the hash-chain and the Merkle-tree.

The source sends the open session packet in SDTP$^+$

1. let $(h_4, h_3, h_2, h_1, h_{\text{root}}) = (H(K_{\text{ack}}), H(h_4), H(h_3), H(h_2), H(h_1))$ in
2. let $(K_{\text{ack}}^1, K_{\text{ack}}^2, K_{\text{ack}}^3, K_{\text{ack}}^4) = (K(1, K_{\text{ack}}), K(2, K_{\text{ack}}), K(3, K_{\text{ack}}), K(4, K_{\text{ack}}))$
3. in let $(K_{\text{ack}}^1, K_{\text{ack}}^2, K_{\text{ack}}^3, K_{\text{ack}}^4) = (H(K_{\text{ack}}^1), H(K_{\text{ack}}^2), H(K_{\text{ack}}^3), H(K_{\text{ack}}^4))$ in
4. let $(S_1, S_2) = (H(K_{\text{ack}}^1, K_{\text{ack}}^2), H(K_{\text{ack}}^3, K_{\text{ack}}^4))$ in let $S_{\text{root}} = H(S_1, S_2)$ in
5. let $\text{sig}_{\text{src}} = \text{sign}(s, d, sID, 5, h_{\text{root}}, S_{\text{root}}, sk_{\text{src}})$ in
6. $\text{sign}(s, d, sID, 5, h_{\text{root}}, S_{\text{root}}, sk_{\text{src}})$.

The hash-chain of length 5 is computed in the first row. Note that the process in the first row is the shorthand (syntax sugar) of the process let $h_4 = H(K_{\text{ack}})$ in let $h_3 = H(h_4)$ in let $h_2 = H(h_3)$ in let $h_1 = H(h_2)$ in let $h_{\text{root}} = H(h_1)$. In the second row, the NACK keys for each packet are generated, based on the NACK master key and the sequence numbers. The leaves of the Merkle-tree are computed by hashing the per-packer NACK keys in the third row. In the fourth row the two first level nodes and the root value are generated. Finally, in rows 5-6, the signature is computed on the open session packet, which is sent to the intermediate node.

After receiving the ACK packet for the open session packet on the channel $c_{\text{siACK}}$, the source starts to send data packets (on the channel $c_d$), which contains the message part and the MAC computed on the message. This part is specified in the similar way as in the SDTP protocol, hence, I omit to discuss it in details. Instead I turn to discuss the case when an ACK packet is received:

The source received an ACK packet in SDTP$^+$

7. $c_{\text{siACK}}(= 4, x_{\text{ackauthval}})$. let $(h_{c}^c, h_{c}^d, h_{c}^e, h_{\text{root}}^c) = (H(x_{\text{ackauthval}}), H(h_{c}^c), H(h_{c}^d), H(h_{c}^e))$ in
8. $[h_{c}^c = h_{\text{root}}] \text{DeleteCacheEntries}(4);$
9. $[c_{\text{siACK}}(= 3, x_{\text{ackauthval}})$. let $(h_{c}^c, h_{c}^c, h_{\text{root}}^c) = (H(x_{\text{ackauthval}}), H(h_{c}^c), H(h_{c}^c))$ in
10. $[h_{c}^c = h_{\text{root}}] \text{DeleteCacheEntries}(3);$
11. $[c_{\text{siACK}}(= 2, x_{\text{ackauthval}})$. let $(h_{c}^c, h_{c}^c, h_{\text{root}}^c) = (H(x_{\text{ackauthval}}), H(h_{c}^c))$ in
12. $[h_{c}^c = h_{\text{root}}] \text{DeleteCacheEntries}(2);$
13. $[c_{\text{siACK}}(= 1, x_{\text{ackauthval}})$. let $h_{c}^c = H(x_{\text{ackauthval}})$ in
14. $[h_{c}^c = h_{\text{root}}] \text{DeleteCacheEntries}(1);$

In this process, four cases are examined according to the sequence number referred to by the ACK message: when the received ACK packet refers to the sequence number $i$, $i \in \{1, 2, 3, 4\}$, the source hashes the received ACK authentication value $i$ times, and compares the result with the stored $h_{\text{root}}$. In case the two values are equal, the source proceeds to delete the cache entries that contains the packets with sequence numbers less than $i$. This is specified in process $\text{DeleteCacheEntries}(i)$.

When specifying the source’s behavior after receiving a NACK packet, within each of the four cases of the ACK message, I further examine different scenarios regarding the values of the sequence numbers of the packets to be retransmitted. Basically, the NACK case can be specified by the consecutive evaluation of the if constructs.

In SDTP$^+$, when the intermediate nodes receive an open session packet on the public channel $c_{\text{idOPEN}}$, they verify the attached signature with public key of the source, then, they store the packet and forward it towards the destination on the public channel $c_{\text{idOPEN}}$. The elements of the open session packet are placed in the $k$-th entry, $e_{k}^d = (s, d, sID, 5, h_{\text{root}}, S_{\text{root}})$, such that the fifth and sixth elements of $e_{k}^d$ store the hash-chain and the Merkle-tree roots, respectively: $5(e_{k}^d) = h_{\text{root}}$ and $6(e_{k}^d) = S_{\text{root}}$. 

92
The intermediate node received the open session packet

15. \( c_{si\text{OPEN}}(x_s, x_d, x_{sID}, x_{m+1}, x_{\text{root}}^\text{hash}, x_{\text{root}}^\text{tree}, x_{\text{sig}}). \ [\text{checksign}(x_{\text{sig}}, pk_{\text{src}}) = ok] [e_k^{si} = \otimes] \)

16. \( \text{let } e_k^{si} = (x_s, x_d, x_{sID}, x_{m+1}, x_{\text{root}}^\text{hash}, x_{\text{root}}^\text{tree}) \text{ in } c_{si\text{OPEN}}((x_s, x_d, x_{sID}, x_{m+1}, x_{\text{root}}^\text{hash}, x_{\text{root}}^\text{tree}, x_{\text{sig}})) \)

The crypt\text{prob} processes that specify the cases when the intermediate node receives an ACK and a NACK packet are received, can be specified in the similar concept to the corresponding processes of the source. The only difference is that the intermediate node forwards the ACK and NACK packets to the source on the channels \( c_{si\text{ACK}} \) and \( c_{si\text{NACK}} \), respectively.

When the destination receives the open session packet, it verifies whether the signature is valid. If so, the hash-chain and the Merkle-tree are generated in the same way as the source (points 1-4). The signature verification can be specified like in point 15 without the if process \( [e_k^{si} = \otimes] \), while the hash-chain and the Merkle-tree generation can be defined like in points 1-4. I continue with specifying the behavior of the destination when it sends an ACK and a NACK. Let denote the tuple of variables \( (x_s, x_d, x_{sID}, x_{apID}, x_{sq}, x_{ear}, x_{rtx}, x_{mac}) \) by datapckvar.

The destination node sends an ACK packet

17. \( c_{sd}(\text{datapckvar}). \ [\text{checkmac}(x_{mac}, K_{sd}) = ok] [nackNbr = 0] [x_{ear} = 1] \text{ UpdateVariables()} \)
18. \[ \text{[ackNbr = 1]. } c_{si\text{ACK}}((1, h_1)). Dst(e_1^{s}-3, \text{ackNbr}, \text{nackNbr, toRTX1, nxtsq}) \]
19. \[ \text{[ackNbr = 2]. } c_{si\text{ACK}}((2, h_2)). Dst(e_1^{s}-3, \text{ackNbr}, \text{nackNbr, toRTX1, nxtsq}) \]
20. \[ \text{[ackNbr = 3]. } c_{si\text{ACK}}((3, h_3)). Dst(e_1^{s}-3, \text{ackNbr}, \text{nackNbr, toRTX1, nxtsq}) \]
21. \[ \text{[ackNbr = 4]. } c_{si\text{ACK}}((4, h_4)). Dst(e_1^{s}-3, \text{ackNbr}, \text{nackNbr, toRTX1, nxtsq}) \]

In rows 17, after receiving the data packet with sequence number \( x_{sq} \) and the EAR bit, \( x_{ear} = 1 \), the destination checks the MAC with the key \( K_{sd} \) that it shares with the source. If \( \text{nackNbr} = 0 \), that is, the number of missing packets is zero, then the destination sends the ACK packet according to the number of the packets received so far (points 18-21). In the process \text{UpdateVariables()}(\), the value of variables, such as \( \text{nackNbr} \) and \( \text{ackNbr} \), are updated. In case \( \text{nackNbr} \) is greater than 0, the destination composes and sends the NACK packet. Within each value of \( \text{ackNbr} \), the destination sends NACK packets on channel \( c_{sdN\text{ACK}} \) according to the gaps between two received packets.

Finally, in ProtSDTP\text{plus(params)}(\), process \text{Dst} and process \text{Int} output the predefined constant \text{BadOpen} on channel \( c_{sd\text{OPEN}} \) when the verification of the signature computed on the open session packet fails. Process \text{Dst} outputs the constant \text{BadData} on \( c_{badPCK} \) when the verification of the MAC computed on a given data packet fails. After receiving an ACK or a NACK, the processes \text{Src} and \text{Int} output the constant \text{BadControl} on \( c_{badPCK} \) when the verifications of the hash-chain and Merkle-tree roots fail.

3.7 Security analysis of WSN transport protocols using crypt\text{prob} time

In my formal proofs, I apply the proof technique that is usual in process algebras. Namely, I define an ideal version of the protocol run, in which I specify the ideal/secure operation of the real protocol. This ideal operation, for example, can require that honest nodes always know what is the correct message they should receive/send, and always follow the protocol correctly, despite the presence of attackers. Then, I examine whether the real and the ideal versions, running in parallel with the same attacker(s), are weak prob-timed bisimilar.

**Definition 16.** Let the processes \text{Prot()} and \text{Prot}^{\text{ideal}}() specify the real and ideal versions of some protocol \text{Prot}, respectively. We say that \text{Prot} is secure (up to the strictness of the ideal version) if \text{Prot()} and \text{Prot}^{\text{ideal}}() are weak probabilistic timed bisimilar: \text{Prot()} \approx_{\text{pt}} \text{Prot}^{\text{ideal}}().
The strictness of the security requirement, which we expect a protocol to fulfill, depends on how ideally/securely we specify the ideal version. Intuitively, Definition 16 says that Prot is secure if the attackers, who can observe the output messages on public channels, cannot distinguish the operation of the two instances.

![Figure 3.3: The difference between the real and ideal version of the DTSN and the SDTP protocols.](image)

The main difference between the ideal and the real systems is that in the ideal system, honest nodes are always informed about what kind of packets or messages they should receive from the honest sender node. This can be achieved by defining hidden or private channels between honest parties, on which the communication cannot be observed by attacker(s). In Figure 3.3 we show the difference in more details. In the ideal case, three private channels are defined which are not available to the attacker(s). Src, Int and Dst denote the processes for the source, the intermediate and the destination nodes. Channels \( c_{\text{privSD}} \), \( c_{\text{privID}} \) and \( c_{\text{privSI}} \) are defined between processes Src and Dst, Int and Dst, Src and Int, respectively. In the rest of the dissertation, I refer to the source, intermediate and destination nodes as S, I and D, respectively. Whenever S sends a packet \( pck \) on public channel \( c_{\text{si}} \), it also informs I about what should I receive, by sending at the same time \( pck \) directly via private channel \( c_{\text{privSI}} \) to I, so when I receives a packet via \( c_{\text{si}} \) it compares the message with \( pck \). The same happens when I sends a packet to D. Whenever, a honest node receives an unexpected data, it interrupts its normal operation. The channels \( c_{\text{privSD}} \) and \( c_{\text{privID}} \) can be used by the destination to inform S and I about the messages to be retransmitted. I recall that the communication via a private channel is not observable by the environment, hence, it can be seen as a silent \( \tau \) transition. Note that for simplicity, I omitted to include the upper layer and channel \( c_{\text{sup}} \) in the Figure 3.3, but I put them in my specification. Finally, I also add additional public channels \( c_{\text{emptyC}} \) for signalling that the cache has been emptied at a given node. These additional channels are defined only for applying bisimilarities in the security proofs, but they do not affect the correctness of the protocol.

With this definition I ensure that the source and intermediate nodes are not susceptible to the modification or forging of \( \text{ACK} \) and \( \text{NACK} \) messages since they make the correct decision either on retransmitting or deleting the stored packets. Namely, this means that the honest nodes only handle the messages received on public channels when they are equal to the expected messages received on private channels.

The attacker model \( M_A \): I assume that an attacker can intercept the information output by the honest nodes on public channels, and modify them according to its knowledge and computation ability. The attacker’s knowledge consists of the intercepted outputs during the protocol run and the information it can create. The attacker(s) can modify the elements of the plaintexts, such as the base number and the bits of the \( \text{ACK} / \text{NACK} \) messages, the EAR and RTX bits and sequence number in data packets. The attacker can also create entire data or control packets including data it possesses. Further, attacker(s) can send packets to its neighborhood. I also assume several attackers who can share information with each other.

To describe the activity of the attacker(s), I apply the concept of the environment, used in the applied \( \pi \)-calculus [27] that models the presence of the attacker(s) in an implicit way. Every message that is output on a public channel is available for the environment, that is, the environment can be seen as a group of attackers who can share information with each other, for instance, via a
3.7. Security analysis of DTSN

The security properties I want to check in case of the DTSN protocol is that how secure it is against the manipulation of control and data packets. In particular, can the manipulation of packets prevent DTSN from achieving its design goal. In this section I demonstrate how to formally prove the security or vulnerability of DTSN using crypt\textsubscript{time}.

First of all, I assume that in both DTSN and SDTP, each action (verification, sending, receiving on public channel) takes an equal amount of time \(d\), and the function \(f\) in Definition 15 returns \(\sum d_i\). This assumption does not change the correctness of the protocols. I define the ideal version of the process \(\text{Prot}(\text{params})\), denoted by \(\text{Prot}^\text{ideal}(\text{params})\), which contains the ideal version of \(\text{DTSN}(\text{params})\):

\[
/* \text{The ideal version of the DTSN protocol for the given topology} */
\text{Prot}^\text{ideal}(\text{params}) \triangleright= \begin{cases}
\text{INIDTSN}^\text{ideal}(i)
\end{cases}
\]

where process \(\text{INIDTSN}^\text{ideal}(i)\) contains \(\text{DTSN}^\text{ideal}(\text{params})\) instead of \(\text{DTSN}(\text{params})\).

To prove or refute the bisimilarity relation, I define \(\text{Prot}(\text{params})\) and \(\text{Prot}^\text{ideal}(\text{params})\) such that the source and intermediate nodes output the constants \(\text{CacheEmptyS}\) and \(\text{CacheEmptyI}\) on the public channel \(\text{emptycacheC}\), respectively, whenever they have emptied their buffers after processing an ACK or a NACK message. This is defined by the following crypt\textsubscript{time} code fragment (for \(i \in \{1, 2, 3\}\)), where process \(\text{checkEi}\) corresponds to the \(i\)-th cache entry:

\[
\text{checkEi}(s, d, \text{apID}, e_{i-3}, sID, \text{acknum}) \triangleq \begin{cases}
\text{emptycacheS} = 3, & \text{emptycacheI} = 3, \text{inc(nbrEcacheS)}
\end{cases}
\]

In case of the intermediate node \(\text{nbrEcacheI}\) and \(\text{CacheEmptyI}\) are in the places of \(\text{nbrEcacheS}\) and \(\text{CacheEmptyS}\). The constants \(\text{CacheEmptyS}\) and \(\text{CacheEmptyI}\) are output whenever the number of the empty cache entries, \(\text{emptycacheS}, \text{emptycacheI}\) is 3, which means that the buffers of \(S\) and \(I\) are emptied, respectively.

**Lemma 4.** With the defined attacker model \(\mathcal{M}_A\), the DTSN protocol is insecure against message manipulation attacks.

According to Definition 15 processes \(\text{Prot}(\text{params})\) and \(\text{Prot}^\text{ideal}(\text{params})\) are not weak probedisimilar because each point of the definition is violated. The following proof show that DTSN is vulnerable to the manipulation of control packets: in SC-1 the attacker increases the base number in ACK packets causing that the stored packets are deleted from the cache although they should not be, while in SC-2 the attacker forces the destination node to send unnecessary ACKs or NACKs.
3. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WSN TRANSPORT PROTOCOLS

- **Scenario SC-1**: This scenario occurs in the topology $S-I$ that includes the attacker $A$ within the transmission range of both $S$ and $I$, which considers also the case $S-A-I$ when $A$ is an internal attacker. I show that there is a trace of probabilistic timed transitions in the real system, denoted by $\text{PTTR}_{\text{real}SC_1}$, which cannot be simulated with any corresponding trace in the ideal system. The trace $\text{PTTR}_{\text{real}SC_1}$ describes the scenario where the source sends the first packet (with sequence number 1) to the intermediate node on the channel $c_{si}$. Because $c_{si}$ is public, this packet is obtained by the attacker(s) (i.e., the environment), who instead of forwarding it, sends an $\text{ACK}$ with the acknowledgement number 1 to $S$. $S$ received this message on $c_{sale}ACK$, empties its buffer and outputs the constant $\text{CacheEmptyS}$ on the public channel $c_{emptyC}$. Since $\text{CacheEmptyS}$ will not be output in $\text{Prot}_{\text{ideal}}$ if node $I$ has not sent anything, the first point of Definition 15 is violated.

- **Scenario SC-2**: I prove that DTSN is susceptible to the attacks that cause futile energy consumption, by showing the violation of the second point of Definition 15. This scenario can happen in the topology $S-I-D$ that includes the attacker $A$ within the transmission range of both $I$ and $D$. In this setting, $A$ can be either an external or an internal attacker, where in the latter case we have the topology $S-I-A-D$. The following trace in the real system cannot be simulated in the ideal system: node $I$ sends a correct packet towards $D$, including the $\text{EAR}$ flag equal to 0, which is intercepted/received by $A$. Then $A$ forwards the packet to $D$, but set the $\text{EAR}$ flag in it to 1, requiring $D$ to send an $\text{ACK}$ or a $\text{NACK}$.

Let the series of silent transitions $s \xrightarrow{\tau(d), \pi} s'$, denoted by $\text{PTTR}_{\text{real}SC_2}$, describe the verification steps made by $D$ after receiving this incorrect packet from $A$. Although at this time there is not any difference in the message outputs (i.e., the frames of the ideal and real systems), the ideal system still cannot simulate $\text{PTTR}_{\text{real}SC_2}$, because according to the definition of $\text{Prot}_{\text{ideal}}(\text{params})$, after receiving the incorrect packet, $D$ performs only one equality checking step between the received and the expected packet received in $c_{id}$ and $c_{privID}$, which takes less time units than in case of the real system. In the real system, node $D$ has to examine the number of missing packets, then, compose and send back a $\text{ACK}$ or an $\text{NACK}$ packet, which consume more time.

3.7.2 Security analysis of SDTP

The ideal version of process $\text{ProtSDTP}(\text{params})$, denoted by $\text{ProtSDTP}_{\text{ideal}}(\text{params})$, is composed of the same specification of processes $\text{Src}$, $\text{Int}$, $\text{Dst}$ as $\text{ProtSDTP}(\text{params})$. The only difference is that in $\text{ProtSDTP}_{\text{ideal}}(\text{params})$, the processes $\text{Src}$ and $\text{Int}$ are defined such that whenever the MAC verification made by $S$ and $I$ on the received $\text{ACK}$/$\text{NACK}$ message fails or an unexpected $\text{ACK}$/$\text{NACK}$ message is received (i.e., the received message does not equal to the $\text{ACK}$/$\text{NACK}$ that has been sent on the hidden channels $c_{privS1}, c_{privID}$), $S$ and $I$ output the predefined constant $\text{BadControl}$ via the public channel $c_{badPKC}$. Similarly, when the destination received an incorrect or unexpected data packet it outputs $\text{BadData}$ on $c_{badPKC}$. Note that this extension does not affect the correctness of SDTP, and only plays a role in the proofs of weak prob-timed bisimilarities.

Since the main purpose of SDTP is using cryptographic means to patch the security holes of DTSN, I examine the security of SDTP according to each discussed attack scenario to which DTSN is vulnerable.

- **Scenario SC-1**: I prove that SDTP is not vulnerable to the attack scenario (SC-1) by showing that $\text{ProtSDTP}_{\text{ideal}}(\text{params})$ can simulate (according to Definition 15) the transition traces produced by $\text{ProtSDTP}(\text{params})$. In SDTP the packet sent by $S$ includes the ACK MAC and $\text{NACK}$ MAC. Hence, when the attacker $A$ sends the $\text{ACK}$ to $S$, in both the real and the ideal systems the source node outputs the constant $\text{BadControl}$ on the channel $c_{badPKC}$, either because the MAC verification fails (in the real system) or the received packet is not the expected one (ideal system). Recall that the MAC verification fails because the attacker does not posses the $\text{ACK}$/$\text{NACK}$ keys of the source. Assume that the Definition 15 holds for the real and ideal systems holds before this trace. According to the first point of
therefore, the attacker model $\mathcal{M}_1$. 

Lemma 5. The SDTP protocol is insecure besides the attacker model $\mathcal{M}_1$.

To prove the vulnerability of SDTP using the prob-timed bisimilarity, I modify (relax) the definition of the ideal model such that the honest nodes in the ideal system only compare the received messages of the destination node with the expected messages.

Scenario SC-2: Now I prove that SDTP is not vulnerable to the attack scenario (SC-2), too. For the transition trace $PTTR_{realSC_2}$ produced by $ProtSDTP^{\text{params}}$, there is a corresponding trace $PTTR_{idealSC_2}$ in the ideal system that simulates $PTTR_{realSC_2}$. After receiving an incorrect packet with $EAR$ bit set to 1, in both the real and the ideal systems the destination node outputs the constant $BadControl$ on the channel $c_{badPK}$, either because the MAC verification fails (in the real system) or the received message is not the expected data packet that has been informed to $I$ on channel $c_{privID}$ (in the ideal system). Hence, both systems consume equal time units during $PTTR_{realSC_2}$ and $PTTR_{idealSC_2}$, which fulfills the second and third points of Definition 15. The first point of the definition also holds because during corresponding traces, systems can output the same data.

As we can see, with the security extensions SDTP could eliminate the essential weaknesses of DTSN, however, I will prove that it is still vulnerable, by showing a trace in the real system $ProtSDTP^{\text{params}}$, which cannot be simulated in $ProtSDTP^{\text{ideal}}^{\text{params}}$.

3.7.3 Security analysis of SDTP+

In the ideal version of process $ProtSDTPplus^{\text{params}}$, denoted by $ProtSDTPplus^{\text{ideal}}^{\text{params}}$, the specification of the processes $Src$, $Int$, and $Dst$ are extended with some additional equality checks between the messages sent on the corresponding private and public channels. Specifically, processes $Dst$ and $Int$ output the predefined constant $BadOpen$ on channel $c_{badOPEN}$ when they receive an
3. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WSN TRANSPORT PROTOCOLS

unexpected open session packet. Namely, when the message which they receive on the public channels $c_{\text{OPEN}}$ and $c_{\text{DOOPEN}}$ is not equal to the corresponding correct open session packet are received on the private channels $c_{\text{prevSI}}$ and $c_{\text{prevID}}$, respectively. Process $\text{Dst}$ outputs the constant $\text{BadData}$ on $c_{\text{badPCK}}$ when it receives an unexpected data packet. Finally, after receiving an unexpected $\text{ACK}$ or a $\text{NACK}$, the processes $\text{Src}$ and $\text{Int}$ output the constant $\text{BadControl}$ on $c_{\text{badPCK}}$. Similarly to SDTP, the attack scenarios SC-1 and SC-2 do not work in SDTP$^+$:

- **Scenario SC-1**: SDTP$^+$ is not vulnerable to the attack scenario (SC-1) because process $\text{ProtSDTPplus}^{\text{ideal}}(\text{params})$ can simulate (according to Definition 15) the transition traces produced by $\text{ProtSDTPplus}(\text{params})$. In SDTP$^+$, $S$ verifies the $\text{ACK}$ and the $\text{NACK}$ packets by comparing the stored roots of the hash-chain and the Merkle-tree with the re-computed roots. Because I did not define any equation for the hash function $H(t)$, from $H(h_i)$ the value of $h_i$ cannot be derived.

Assume that the source are storing the first three packets for a given session. The buffer of the source will be emptied only when the $\text{ACK}$ packet, $\text{ACK} = (m, h_m)$, is received such that $m = 3$. This is because the root of the hash-chain, $h_{\text{root}} = H(H(H(K_{\text{ack}})))$, and based on the fact that $H(t_1) = H(t_2)$ if and only if $t_1 = t_2$, the $m$-time hashing on $h_m$ must be $H(H(H(H(K_{\text{ack}}))))$. To empty the buffer, $m$ must be at least 3. In case $m = 4$, $h_m$ must be $H(K_{\text{ack}})$. This hash value cannot be computed by the attackers because the source and the destination never reveals $K_{\text{ack}}$. Hence, the attackers must receive or intercept $H(K_{\text{ack}})$ from a honest node, which means that $H(K_{\text{ack}})$ has been revealed by the destination. $m$ cannot be greater than four, otherwise, the attackers must have $K_{\text{ack}}$, or the destination must revealed $K_{\text{ack}}$, which according to the protocol, will never happen. When $m = 3$, $h_m$ must be $H(H(K_{\text{ack}}))$, which cannot be computed by attacker nodes. Therefore, either the attacker sends a correct $\text{ACK}$ or the $\text{ACK}$ with incorrect authentication value, the ideal and the real systems can simulate each other. In the first case, the constant $\text{EmptyCacheS}$, while in the second case $\text{BadControl}$ is output in both systems.

- **Scenario SC-2**: The proof regarding the scenario SC-2 in case of SDTP$^+$ is bit complicated, because, in SDTP$^+$ additional timers and restrictions [Th11 , 2013] are used in order to limit the destination to send back an $\text{ACK}$ or a $\text{NACK}$, after receiving a data packet in which the attacker modified the EAR flag to 1. I have to relax the definition of weak prob-timed labeled bisimilarity, and modify the specification of the destination in the ideal system. The detailed discussion can be found in the section 7 of my report [Th12 , 2013].

- **Scenario SC-3**: This scenario examine whether the attackers can make the intermediate node incorrectly empty its buffer. Let us assume that $I$ has already accepted the open session packet and has stored the hash-chain root, denoted by $h_{\text{root}}$, in it. The signature in the open session packet must be computed with the secret key of the source, $s_{\text{src}}$. This is because after receiving a packet on channel $c_{\text{OPEN}}$, process $\text{Int}$ performs the verification $\text{checksign}(x_{\text{src}}, pk_{\text{src}}) = ok$, and only the signature computed with $sk_{\text{src}}$ can be verified with $pk_{\text{src}}$. However, this means that $h_{\text{root}}$ must be generated by the source, that is, $h_{\text{root}} = H(H(H(H(K_{\text{ack}}))))$. Again, assume that in the current state $I$ stores the first three data packets. From this point, the reasoning is similar to the scenario SC-1, namely, either the constant $\text{EmptyCacheI}$ or $\text{BadControl}$ is output in both the ideal and the real systems. Hence, the sandwitch attack does not work in SDTP$^+$.

3.8 Automated security verification using the PAT process analysis toolkit

In this subsection, I propose an automated verification method for WSN transport protocols. My method is based on the application of the widely used PAT process analysis toolkit [68], which provides a very expressive and high-level specification language. PAT has been widely used for
verifying timed and probabilistic systems, but to the best of my knowledge, it has not been used for verifying WSN transport protocols before. Its expressive modeling language makes PAT be the most suitable tool for analyzing WSN transport protocols, because it enables us to specify transport protocols in a most convenient way, compared with many other tools [57], [11], [13], [43]. I also has to mention that with crypts, my purpose is to develop a formal proof method for probabilistic timed cryptographic protocols, and the question of how can an automated verification method based on crypts be designed is left for the future.

3.8.1 My work and the related works

**Related Methods:** The SPIN [57] and the UPPAAL [11] model-checkers are general purpose model-checking tools. CPAL-ES [48], and ProVerif [13] are automatic verification tools developed for verifying security protocols. The main drawback of them is that they lack semantics and syntax for defining the systems that include probabilistic and timed behavior. Hence, they cannot be used to verify WSN transport protocols such as DTSN and SDTP. The PRISM model-checker [43] supports probabilistic and real time systems but its limited specification language does not enable us to verify protocols/systems that perform complex computations which require, for instance, the usage of variables arrays, sending and receiving messages on channels.

**My Method:** My method is based on the PAT process analysis toolkit. PAT [68] is a self-contained framework to specify and automatically verify different properties of concurrent (i.e. supporting parallel compositions construct), timed systems with probabilistic behavior. It provides a user friendly graphical interface, a featured model editor and an animated simulator for debugging purposes. PAT implements various state-of-the-art model checking techniques for different properties such as reachability, LTL properties with fairness assumptions, refinement checking and probabilistic model checking. To handle large state spaces, the framework also includes state-of-the-art optimization methods.

One of the biggest advantage of PAT compared with other solutions is that it supports probabilistic and timed behavioral syntax and semantics, which are important in my case. It contains several modules to deal with problems in different domains including real time and probabilistic systems. PAT has been used to model and verify a variety of systems, such as distributed algorithms, and real-world systems like multi-lift and pacemaker systems. However, PAT (so far) does not provide syntax and semantics for specifying cryptographic primitives and operations, such as digital signature, MAC, encryptions and decryptions, one-way hash function, etc.. Hence, I model cryptographic operations used by SDTP in an abstract, simplified way. Note that the simplication has been made in an intuitive way, based on the definition of a given protocol.

3.8.2 The PAT process analysis toolkit

In this subsection, I briefly introduce the features provided by the main modules of PAT that I use to verify the security of DTSN and SDTP. PAT is basically designed as a general purpose tool, not specifically for security protocols. It provides a CSP [33] like syntax, which is a process calculus for concurrent systems, but it is more expressive than CSP because it also includes the language constructs for time and probabilistic issues. PAT also provides programming elements like communication channels, array of variables and array of channels, similarly to Promela [34] (Process Meta Language), the specification language used by the SPIN [34] model-checker. PAT handles time in a tricky way, namely, instead of modeling clocks and clock resets in an explicit manner, to make the automatic verification more efficient it applies an implicit representation of time (clocks).

**Communicating Sequential Programs (CSP#) Module.** The CSP# module supports a rich modeling language named CSP# (a modified variant of CSP) that combines high-level modeling operators like (conditional or non-deterministic) choices, interrupt, (alphabetized) parallel composition, interleaving, hiding, asynchronous message passing channel.

The high-level operators are based on the classic process algebra Communicating Sequential Processes (CSP). Beside keeping the original CSP as a sub-language, CSP# offering a connection
3. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WSN TRANSPORT PROTOCOLS

to the data states and executable data operations.

Global constant is defined using the syntax

#define constname val

where constname is the name of the constant and val is the value of the constant. Variables
and array can be defined as follows

1. var varname = val;  2. var arrayname = [val_1,..., val_n]; 3. var arrayname[n];

In PAT variables can take integer values. The first point defines the variable with name varname
with the initial value val; the second point defines the fix size array with n values, and the third
point declares the array of size n, where each element is initialized to 0. To assign values to specific
elements in an array, event prefix is used as follows:

P() = assignvalEV{arrayname[i] = val} -> Skip,

where the assignment of the i-th element of the array arrayname is performed within the scope of
the event assignvalEV. Skip is a special process that models termination of a process that contains
it, similar to the nil process in cryptprob.

In PAT, process may communicate through message passing on channels. Channels and out-
put/input actions on a channel can be declared using the syntax below:

1. (declaration of channel channname): channel channname size;
2. (output of the msg tuple (m1,m2,m3) on channname): channname!m1.m2.m3;
3. (input a msg (m1,m2,m3) on the channel channname): channname?x1.x2.x3;

channel is a keyword for declaring channels only, channname is the channel name and size is
the channel buffer size. It is important that a channel with buffer size 0 sends/receives messages
synchronously. A process is a relevant specification element in PAT that is defined as an equation

P(x1, x2, ..., xn) = ProcExp;

where ProcExp defines the behavior of process P. PAT defines special processes to make the coding
more convenient: Process Stop is the deadlock process that does nothing; process Skip terminates
immediately and then behaves exactly the same as Stop.

Events are defined in PAT to make debugging be more straightforward and to make the returned
attack traces be more readable. A simple event is a name for representing an observation. Given
a process P, the syntax ev -> P describes a process which performs ev first and then behaves as
P. An event ev can be a simple event or can be attached with assignments which update global
variables as in the following example, ev{x = x + 1;} -> Stop; where x is a global variable.
PAT supports almost every mathematical operators like in the C programming language, such as
plus, minus, times, division, modulo, negation of boolean variables, etc. PAT also supports many
familiar constructs such as while, case, if-then, and atomic action feature. The assignment attached
to events is a program that may consist of these operations and constructs.

A sequential composition of two processes P and Q is written as P ; Q in which P starts first
and Q starts only when P has finished. A (general) choice is written as P [ | ] Q, which states that
either P or Q may execute. If P performs an event first, then P takes control. Otherwise, Q takes
control. Interleaving represents two processes which run concurrently, and is denoted by P ||| Q.
Process P ||| Q is equivalent to the parallel composition in cryptprob.

Assertion: An assertion is a query about the system behaviors. PAT provides queries for
deadlock-freeness, divergence-freeness, deterministic, nonterminating, reachability, respectively as
in the following syntax:

1. #assert P() deadlockfree; /* asks if P() is deadlock-free or not. */
2. #assert P() divergencefree; /* asks if P() is divergence-free or not. */
3.8. Automated security verification using the PAT process analysis toolkit

3. #assert P() deterministic; /* asks if P() is deterministic or not. */
4. #assert P() nonterminating; /* asks if P() is nonterminating or not. */
5. #assert P() reaches cond; /* asks if P() can reach a state where cond is satisfied. */

PAT’s model checker performs Depth-First-Search or Breath-First-Search algorithm to repeatedly explore unvisited states until a deadlock state (i.e., a state with no further move).

A goal (badstate, goodstate, etc.) is a boolean expression, for example, if we want to define the goal that the value of $x$ is 5, we write the following

```
#define goal (x==5);
```

In PAT the mathematical operations and expressions can be specified in the C like style. PAT supports FDR’s approach for checking whether an implementation satisfies a specification or not.

**Real-Time System (RTS) Module.** The RTS module in PAT enables us to specify and analyze real-time systems and verify time concerned properties. To make the automatic verification be more efficient, unlike timed automata that define explicit clock variables and capturing real-time constraints by explicitly setting/resetting clock variables, PAT defines several timed behavioral patterns are used to capture high-level quantitative timing requirements wait, timeout, deadline, waituntil, timed interrupt, within.

1. **Wait**: A wait process, denoted by Wait[$t$], delays the system execution for a period of $t$ time units then terminates. For instance, process Wait[$t$];$P$ delays the starting time of $P$ by exactly $t$ time units.

2. **Timeout**: Process $P$ timeout[$t$] $Q$ passes control to process $Q$ if no event has occurred in process $P$ before $t$ time units have elapsed.

3. **Timed Interrupt**: Process $P$ interrupt[$t$] $Q$ behaves as $P$ until $t$ time units elapse and then switches to $Q$. For instance, process (ev1 -> ev2 -> ...) interrupt[$t$] $Q$ may engage in event ev1, ev2 ... as long as $t$ time units haven’t elapsed. Once $t$ time units have elapsed, then the process transforms to $Q$.

4. **Deadline**: Process $P$ deadline[$t$] is constrained to terminate within $t$ time units.

5. **Within**: The within operator forces the process to make an observable move within the given time frame. For example, $P$ within[$t$] says the first visible event of $P$ must be engaged within $t$ time units.

**Probability RTS (PRTS) Module.** The PRTS module supports means for analyzing probabilistic real-timed systems by extending RTS module with probabilistic choices and assertions.

The most important extension added by the PRTS module is the probabilistic choice (defined with the keyword `pcase`):

```
prtsP = pcase {
    [prob1] : prtsQ1
    [prob2] : prtsQ2
    ...
    [probn] : prtsQn
};
```

where `prtsP, prtsQ1, ..., prtsQn` are PRTS processes which can be a normal process, a timed process, a probabilistic process or a probabilistic timed process. `prtsP` can proceed as `prtsQ1, prtsQ2, ..., prtsQn` with probability `prob1, prob2, ..., probn`, respectively.
3. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WSN TRANSPORT PROTOCOLS

For user’s convenience, PAT supports another format of representing probabilities by using weights instead of probs in the pcase construct. In particular, instead of prob1, ..., probn we can define weight1, ..., weightn, respectively, such that the probability that prtsP proceeds as prtsQ1 is weight1 / (weight1 + weight2 + ... + weightn).

**Probabilistic Assertions:** A probabilistic assertion is a query about the system probabilistic behaviors. PAT provides queries for deadlock-freeness with probability, reachability with probability, Linear Temporal Logic (LTL) with probability, and refinement checking with probability, respectively as in the following syntax:

1. #assert prtsP() deadlockfree with pmin/ pmax/ prob;
2. #assert prtsP() reaches cond with prob/ pmin/ pmax;
3. #assert prtsP() |= F with prob/ pmin/ pmax;

The first assertion asks the (min/max/both) probability that prtsP() is deadlock-free or not; the second assertion asks the (min/max/both) probability that prtsP() can reach a state at which some given condition cond is satisfied; the third point asks the (min/max/both) probability that prtsP() satisfies the LTL formula F.

3.8.3 Defining the attacker model

The main drawback of PAT is that it is not optimized for verifying security protocols in presence of adversaries, hence, in the current form it does not support a convenient way for modeling attackers. In PAT the attacker(s) are not included by default, and the user has to define the attacker’s behavior and its place in the network explicitly.

In order to reduce the state space during the verification (for preventing run out of memory), I have to restrict the attacker’s ability, instead of allowing the attackers to perform unlimited operations. The restriction are specified according to the messages exchanged between honest nodes in DTSN and SDTP. More specifically, the attacker intercepts every message sent by its neighbors, and it can modify the content of the intercepted packet as follows:

- it can increase, decrease or replace the sequence number in data packets;
- it can set/unset the EAR bit and RTX flag in each data packet;
- it can increase, decrease or replace the base (ack) number in ACK/NACK packets;
- it can change the elements in NACK packets;
- it can include or replace the correct MACs with the self computed MACs (with owned keys).
- the combination of these actions.

Finally, the attacker can forward the modified packets to the neighbor nodes. There can be two attackers who can share keys and cooperate. In addition, I assume that an attacker has no memory, namely, it can construct messages only based on the latest information it receives, or its generated data.

Similarly to the analysis performed with cryptprob time, we assume three honest nodes, namely, the source S, the intermediate node I, and the destination node D. Again, I assume that the source and the destination cannot be an attacker. To analyze the security of DTSN and SDTP, I define the attacker process(es) based on the following scenarios. I examine different places of the attacker(s) in the network: Top1. S − A1 − I − D; Top2. S − I − A2 − D; Top3. S − A1 − I − A2 − D. The attack scenarios that can be detected in these three topologies are also valid in the topologies which include more intermediate nodes.
The behaviors of the attacker nodes $A_1$ and $A_2$ are defined by the processes $\text{procA}_1()$ and $\text{procA}_2()$. In my model, by default the attackers have two sequence number $\text{seqA}_1$ and $\text{seqA}_2$ which are the smallest and the largest possible sequence numbers (i.e., 1 and 4 in my model), respectively. The attackers can include $earA \in \{0,1\}$, $rtxA \in \{0,1\}$ in their data packets. The attackers, in addition, possess the pre-defined values $bA_1,\ldots,bA_4$ for requiring re-transmission in $\text{NACK}$ messages.

Process $\text{procA}_1()$, which defines the behavior of the attacker $A_1$, in Top1 is specified as an external choice among the following four activities, (each of them is composed of additional choice options). For each scenario, I define additional symmetric channels $\text{chASPck}$, $\text{chASAck}$, $\text{chASEAR}$, $\text{chADPck}$, $\text{chADAck}$, $\text{chADNack}$, $\text{chADEAR}$, between the attacker(s) and its(their) honest neighbors.

1. Without receiving any message: (i.) $A_1$ sends a data packet, with $\text{seqA}_1.earA.\text{rtxA}$ or $\text{seqA}_2.earA.\text{rtxA}$, to $I$; (ii.) $A_1$ sends an $\text{ACK}$, for the packet $\text{seqA}_1$ or $\text{seqA}_2$, to $I$ or to $S$; (iii.) $A_1$ sends a $\text{NACK}$, with the ack number $\text{seqA}_1$ or $\text{seqA}_2$, and a combination of $bA_1,\ldots,bA_4$, to $I$ or to $S$.

2. After receiving a data packet ($\text{chSAPck}\text{seq.ear.}\text{rtxA}$): (i.) $A_1$ sends a data packet, with the sequence number $\text{seqA}_1$ or $\text{seqA}_2$, and different values EAR/RTX bits, to $I$; (ii.) $A_1$ sends an $\text{ACK}$, with the ack number $\text{seqA}_1$ or $\text{seqA}_2$, to $I$ or to $S$; (iii.) $A_1$ sends a $\text{NACK}$, with the ack number $\text{seqA}_1$ or $\text{seqA}_2$, and a combination of $bA_1,\ldots,bA_4$, to $I$ or to $S$.

3. After receiving an $\text{ACK}$ (chIAAck?ack): (i.) $A_1$ sends a data packet, with the sequence number $\text{ack}$, $\text{seqA}_1$ or $\text{seqA}_2$, to $I$; (ii.) $A_1$ sends an $\text{ACK}$, with the ack number $\text{ack}$, $\text{seqA}_1$ or $\text{seqA}_2$, to $I$ or to $S$; (iii.) $A_1$ sends a $\text{NACK}$, with the ack number $\text{ack}$, $\text{seqA}_1$ or $\text{seqA}_2$, and a combination of $bA_1,\ldots,bA_4$, to $I$ or to $S$.

4. After receiving a $\text{NACK}$ with 1-4 bits ($\text{chIAnack?ack.b1 [ ] chIAnack?ack.b2 [ ] chIAnack?ack.b1.b2.b3.b4}$): (i.) $A_1$ sends a data packet, with the sequence number $\text{ack}$, $\text{seqA}_1$ or $\text{seqA}_2$, to $I$; (ii.) $A_1$ sends an $\text{ACK}$, with the sequence number $\text{ack}$, $\text{seqA}_1$ or $\text{seqA}_2$, and a combination of $bA_1,\ldots,bA_4$, $b_1$, $b_2$, $b_3$, $b_4$, to $I$ or to $S$; (iii.) $A_1$ sends a $\text{NACK}$ to $I$ or to $S$. I recall that the attacker, besides the self-generated data, can only use the information in the received messages. Hence, when the attacker receives the $\text{NACK}$ $\text{ack.b1}$, it can only use (besides its own data) $\text{ack}$ and $b1$.

For the topology Top2, the attacker process $\text{procA}_2()$ describing the behavior of $A_2$ is specified in the same way as $\text{procA}_1()$, but with different channels. In $\text{procA}_2()$, the PAT codes are the same as in the case of $\text{procA}_1()$ except that the used channels at each corresponding step are changed as follows: In the first topology, $A_1$ receives data packets from $S$ on $\text{chSAPck}$, which is changed to $\text{chIAPck}$ in the second case because now data packets come from $I$. Similarly, the inputs on $\text{chSAAck}$ and $\text{chSANack}$ are changed to $\text{chDAAck}$ and $\text{chDANack}$, respectively. The outputs by $A$ on $\text{chIAPck}$, $\text{chIAAck}$, $\text{chIAAck}$, $\text{chSAAck}$ and $\text{chSANack}$ are changed to $\text{chDAAck}$, $\text{chDAAck}$, $\text{chDANack}$, $\text{chIAAck}$ and $\text{chIAnack}$, respectively.

In case of SDTP, of course, the $\text{ACK}$ and $\text{NACK}$ messages also contain the corresponding per-packet $\text{ACK}$ and $\text{NACK}$ keys, which I will show in detail in the next sections. Finally, I denote the attackers with the behavior specified by the processes $\text{procA}_1()$, $\text{procA}_2()$ and their sub-processes by $M^\text{AT}_A$. The sub-processes of $\text{procA}_1()$ and $\text{procA}_2()$ are denoted by $\text{subA}_1()$ and $\text{subA}_2()$, respectively, and they specify the attackers whose activity is the subset of the activity defined in $\text{procA}_1()$ or $\text{procA}_2()$.

### 3.8.4 On verifying DTSN using PAT

Following the concept in Section 3.7, I define public (symmetric) channels between each node pair. The channels $\text{chSPIpc}$, $\text{chDIpc}$ are for transferring data packet, while $\text{chSIAck}$, $\text{chSINack}$, $\text{chDIack}$, $\text{chDINack}$, $\text{chSIEar}$, and $\text{chDIEar}$ are for $\text{ACK}$, $\text{NACK}$, and EAR messages, respectively.
3. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WSN TRANSPORT PROTOCOLS

In addition, I add channels $chUpSPck$ and $chUpDPck$ between the upper layer and the two nodes $S$ and $D$, respectively. I also define the channel $chEndSession$ between the source and the other entities, for indicating the end of a session.

I define different constants such as the $ERROR$ for signalling errors according to DTSN, and the time values of timers, the size of the buffers (every node has the same buffer size), the maximal value of packet for a session, and the constant $EARPCK$ that represents a stand-alone $EAR$ packet.

I assume that the probability that a packet sent by a node has been lost and does not reach the addressee, denoted by the constant $PLOST$. The probability that an intermediate node stores a packet, denoted by the constant $PSTORE$. The value of activity timer is defined by the constant $TACT$, while the value of the $EAR$ timer is defined by $TEAR$. Finally, I defined the maximal number of $EAR$ attempts with the constant $MAXEAR$. I give each of these constants a value, which are only (intuitively meaningful) examples for building and running the program code.

I assume the buffer of each honest node contains three cache entries, which are declared by 3-element arrays of variables. Numerous global variables are declared for searching in the arrays, as well as keeping track of the empty cache entries, and the next cache entry to be deleted, etc.

I specify the following bad states regarding the three topologies Top1, Top2 and Top3, in the form of assertions and goals in PAT, which represent the insecurity of the protocol, and I run automatic verification to see whether these bad states can be reached.

Let us consider the topologies Top1 and Top2. The first main design goal of DTSN is to provide reliable delivery of packets. Hence, if the attacker(s) $M^{PAT}_{A}$ can achieve that the probability of delivery of some packet in a session (i.e., the probability of the delivery of all packets in a session) is zero, then we say that DTSN is not secure in the presence of the defined adversaries. The assertion, denoted by $violategoal1$, for verifying the security of DTSN regarding this first main goal is the following:

PAT code:

```
#define violategoal1 (OutBufL == 0 && BufI > 0);
#define violategoal2 (OutBufL == 0 && BufI == 0 && numNACK > 0);
```

where the (global) variables $OutBufL$ and $BufI$ are the number of the occupied cache entries at the source and intermediate node, respectively. The variable $numNACK$ represents the number of the packets that are requiring to be retransmitted, namely, the number of the gaps in the data packet stream received by the destination. Hence, the goal $violategoal1$ represents the state when the buffer of the source has been emptied, however, at the same time the buffer of the intermediate node contains data packet(s). The goal $violategoal2$ represents the state in which the cache of $S$ and $I$ are emptied, but at the same time $D$ has not received all of the packets sent by the source yet.

PAT code:

```
A1. #assert DTSNA1() reaches violategoal1;
A2. #assert DTSNA1() reaches violategoal1 with pmax;
A3. #assert DTSNA2() reaches violategoal2;
A4. #assert DTSNA2() reaches violategoal2 with pmax;
```

Process DTSN for each topology:

```
DTSNA1() =
  UpLayer() ||| procS() ||| procA1() ||| procI() ||| procD();
DTSNA2() =
  UpLayer() ||| procS() ||| procI() ||| procA2() ||| procD();
```

Processes $DTSNA1()$ and $DTSNA2()$ define the DTSN protocol for the topologies Top1 and Top2, respectively. Considering the topology Top1, the PAT model-checker with the default settings
returns **Valid** for the assertion (A1), and (A2) I receive the result that the maximum probability of reaching \( \text{violategoal1} \) in \( DTSN() \) is greater than zero. For the topology \( \text{Top2} \), I run PAT with the default settings for (A3) and (A4) and get **Valid** for (A3), and for (A4) the maximum probability of reaching \( \text{violategoal2} \) in \( DTSN() \) is greater than zero. For each assertion (A1) and (A3) an attack scenario is also returned, in which the attacker \( A \) sent back an \( \text{ACK} \) with the acknowledgement number \( \text{seqA2} \), which is larger than all the possible sequence numbers, to the source and the intermediate nodes, respectively. Unfortunately, PAT only returns one attack scenario for the same definition of process \( DTSN() \) and \( M_{PAT}^A \). Hence, to examine the possibilities of certain attack scenarios, I have to tailor the behavior of the attackers for that scenario. The tailored activity of the attackers can be seen as a subset of \( M_{PAT}^A \).

Now let us consider the topology \( \text{Top3} \) that includes two attackers \( A_1 \) and \( A_2 \). I specify the bad states for DTSN, and I run the model-checker to see if these bad states can be reached. Let the number of buffer entries that are freed at node \( I \) after receiving an \( \text{ACK} / \text{NACK} \) message be \( \text{freenum} \), and the number of packets received in sequence by node \( D \) be \( \text{acknum} \). The bad state, denoted by \( \text{violategoal3} \), specifies the state where (\( \text{freenum} > \text{acknum} \)).

\[
\begin{align*}
\text{PAT code:} & \\
& \#define \text{violategoal3} (\text{freenum} > \text{acknum}) \\
& A5. \ #\text{assert} \ DTSN_{subA1subA2}() \ \text{reaches} \ \text{violategoal3} \\
& A6. \ #\text{assert} \ \text{DTSN}_{A1A2}() \ \text{reaches} \ \text{violategoal3}
\end{align*}
\]

In case the process \( DTSN_{A1A2}() \) reaches \( \text{violategoal3} \), it can be seen as a security hole or an undesired property of DTSN, because according to the definition of \( I \), it should always empty at most as many cache entries as the ack number it receives in \( \text{ACK} / \text{NACK} \) messages (i.e., \( \text{acknum} \)).

As already mentioned, PAT only returns one attack trace, and always the same one. Hence, to obtain different attack scenarios I have to modify \( DTSN_{A1A2}() \) by tailoring the ability of the attackers. In particular, these tailored attackers activities are defined by the processes \( \text{subA1} \) and \( \text{subA2} \), which are the subprocceses of \( \text{procA1} \) and \( \text{procA2} \), respectively. I denote \( DTSN_{subA1subA2}() \) as the process which includes different sub-processes of \( \text{procA1}() \) and \( \text{procA2}() \). Then, I examine if there is any \( DTSN_{subA1subA2}() \) that reaches \( \text{violategoal2} \).

First, \( \text{subA1}() \) for the attacker \( A_1 \) is defined such that 1.) \( A_1 \) sends a data packet to \( I \) without receiving any message (this is defined by process \( A1NotRcvSndPck2I() \)); or after receiving a data packet \( \text{seq.ear.rtx} \) from \( S \), 2.) \( A_1 \) sends a data packet to \( I \) (defined by process \( A1RcvPckSndPck2I() \)); or 3.) \( A_1 \) forwards the packet unchanged to \( I \). The second attacker \( A_2 \) is defined by \( \text{subA2}() \) such that 1.) \( A_2 \) sends an \( \text{ACK} \) to \( I \) without receiving any message (defined by process \( A2NotRcvSndAck2I() \)); or 2.) after receiving a data packet from \( I \), \( A_2 \) sends an \( \text{ACK} \) to \( I \) (defined by process \( A2RcvPckSndAck2I() \)). The PAT code of the two sub-processes is as follows:

\[
\begin{align*}
\text{subA1}() = \\
& A1NotRcvSndPck2I() \\
& \quad \left[ \begin{array}{l}
\text{chSAPck?seq.ear.rtx} \rightarrow \\
\quad \left( \begin{array}{l}
\text{A1RcvPckSndPck2I()} \left[ \begin{array}{l}
\text{chIAPck!seq.ear.rtx} \rightarrow \text{subA1}()
\end{array}\right)
\end{array}\right)
\end{array}\right]
\end{align*}
\]
After running the PAT model-checker for the assertion (A5), the tool returned Valid along with the following scenario:

1. A1 gets the first data from S, and changes the seq number 1 to seqA2
2. A1 sends I the packet seqA2.ear rtx (chIAPck!seqA2.ear rtx)
3. I stores this packet and forwards it to A2 (chIAPck!seqA2.ear rtx)
4. A2, after obtaining this packet (chIAPck?seqA2.ear rtx), sends to I the ACK with the ack number seqA2 (chIAAck!seqA2)
5. I erases its entire buffer, while D has not received any data yet.

As the result, basically, the attackers A1 and A2 can always achieve that the buffer of I is emptied, because by definition, seqA2 is the largest possible sequence number, hence will be larger than every seq number stored by I. In the worst case, node I is always prevented from caching packets which corrupts the design goal of DTSN. I give the name sandwith type attack for this scenario. Note that for the assertion (A6), due to the complexity of the attackers in DTSNA1A2(), the model-checker returns Valid after a larger amount of time and with a much longer attack trace. This long trace essentially shows the same type of attack as the shorter one, detected in the first assertion.

### 3.8.5 On verifying SDTP using PAT

As already mentioned earlier, PAT does not support language elements for specifying cryptographic primitives and operations in an explicit way. I specify the operation of SDTP with the implicit representation of MACs and ACK/NACK keys. First, recall that in SDTP the per-packet ACK and NACK keys are generated as

\[
K^{(n)}_{ACK} = \text{PRF}(K_{ACK}; \text{“per packet ACK key”; } n)
\]

\[
K^{(n)}_{ACK} = \text{PRF}(K_{ACK}; \text{“per packet NACK key”; } n).
\]

Following this concept, in PAT I define the ACK key and NACK key for the packet with sequence number n by the “pair” n.Kack and n.Knack, respectively. To reduce the verification complexity I made abstraction on the key generation procedure, and model the session ACK/NACK master keys by the unique constants Kack and Knack. Then I specify the packets sent by the source node as follows: sq.ear rtx sq sq.Kack sq sq.Knack, where the first part sq.ear rtx contains the packet’s sequence number, the EAR and RTX bits, respectively; the second part sq sq.Kack and the third part sq sq.Knack represent the ACK MAC and NACK MAC computed over the packet with sequence number sq without the EAR and RTX bits, using the per-packet ACK and NACK keys sq.Kack, and sq.Knack. An ACK message has the following forms: acknbr.acknbr.Kack, where acknbr.Kack is the corresponding ACK key of acknbr. A NACK message has the format acknbr.nckb1.acknbr.Kack.nckb.Knack, where nckb.Knack is the NACK key of the packet to be retransmitted. The NACK message can include more bits, in a similar way.

By default, the attackers do not posses the two master keys Kack and Knack of honest nodes, but only their own key Katt. Because honest nodes are specified to wait for these MACs format, the attackers should compose the MACs in this format as well, namely, sq.A sq.A.Katt. The attackers cannot use the master keys to construct the per-packet ACK/NACK keys, and when they obtain a MAC, e.g., sq sq.Kack, they cannot use sq.Kack, only in case they receives sq.Kack.

The SDTP protocol with the first topology is specified as the parallel compositions of each honest node and the attacker A1:

\[
\text{SDTPA1}() = \text{UpLayer}() ||\| \text{procS()} ||\| \text{procA1()} ||\| \text{procI()} ||\| \text{procD}();
\]
Again, each process is recursively called, in such a way that is equivalent to replication of many instances of the processes. The SDTP protocol with the second topology is specified as the following parallel compositions:

\[
\text{SDTPA2}() = \text{UpLayer()} ||| \text{procS()} ||| \text{procI()} ||| \text{procA2()} ||| \text{procD();}
\]

I examine the security of SDTP according to each discussed attack scenario to which DTSN is vulnerable. Specifically, the PAT model-checker is run for the same three goals defined in case of DTSN. The following PAT code is applied for asking if these bad states can be reached in SDTP:

**PAT code:**

B1. \#assert SDTPA1() reaches \text{violategoal1};
B2. \#assert SDTPA1() reaches \text{violategoal1} with \text{pmax};
B3. \#assert SDTPA2() reaches \text{violategoal2};
B4. \#assert SDTPA2() reaches \text{violategoal2} with \text{pmax};

Not like in DTSN, PAT returns Not Valid for the assertion (B1) and (B3). This means that in the presence of the defined attacker $M_{\text{PAT}}^A$, thanks to its security mechanism, SDTP cannot be corrupted such that $D$ has not received some packets and required for retransmissions, but the buffers of $S$ and $I$ are emptied (B3). In addition, the attacker cannot achieve that the source empties its buffer but the intermediate node does not (B1). Running PAT for (B2) and (B4), I get that the maximum probability of reaching violategoal1 and violategoal2 are zero.

Now let us consider the topology $\text{Top3}$ that includes two attackers $A1$ and $A2$. The SDTP protocol with the third topology is specified as the parallel compositions of each honest node and the two attackers:

\[
\text{SDTPA1A2}() = \text{UpLayer()} ||| \text{procS()} ||| \text{procA1()} ||| \text{procI()} ||| \text{procA2()} ||| \text{procD();}
\]

I specify the bad states for SDTP in the same way as in DTSN, with violategoal3. Then I verify whether violategoal3 can be reached in SDTP in the assertions (B5-B6).

B5. \#assert SDTPsubA1subA2() reaches violategoal3;
B6. \#assert SDTPA1A2() reaches violategoal3;

Similar to the case of DTSN, I also run the PAT model-checker with the attacker sub-processes subA1() and subA2(), to examine the existence of the sandwich type attack against DTSN:

subA1() =
/* A1 sends a data packet to I, without receiving any message, OR */
A1NotRcvSndPck2I()
/* After receiving a data packet on channel chSAPck */
[] chSAPck?seq.ear.\text{rtx.seq1.seq2.Kack.seq3.seq4.Knack} ->
( /* A1 sends a data packet to I, OR */
A1RcvPckSndPck2I()
/* A1 forwards the packet unchanged to I */
)

subA2() =
/* A2 sends a data packet to I, without receiving any message, OR */
A2NotRcvSndAck2I()
/* After getting a data on channel chIAPck, A2 sends ACK with seqA2 to I */
3. FORMAL AND AUTOMATED SECURITY VERIFICATION OF WSN TRANSPORT PROTOCOLS


SDTPsubA1subA2() =
  UpLayer() ||| procS() ||| subA1() ||| procI() ||| subA2() ||| procD();

As a result, the tool returned Valid for the assertions B5 and B6 along with the following trace:
1. A1 sends to I a data pck seqA2.ear.rtx with the corresponding ACK MACs: seqA2.seqA2.Katt and NACK MACs: seqA2.seqA2.Katt
2. I stores this pck and forwards it (unchanged) to A2.
3. A2 received this packet and sends to I the ACK for seqA2: seqA2.seqA2.Katt.seqA2.Katt with the 2 keys seqA2.Katt and seqA2.Katt
4. As result, I deletes all the packets stored in its buffer because the key seqA2.Katt and the MAC seqA2.seqA2.Katt match.

In summary, I get the result that both DTSN and SDTP are susceptible for this sandwich style attack scenario. The main reason for this weakness is that in SDTP the intermediate nodes do not verify the origin of the received messages, they only check whether the stored ACK/NACK MACs match the received ACK/NACK keys.

3.9 Summary

In this chapter, I addressed the problem of formal and automated security verification of WSN transport protocols that may perform cryptographic operations. The verification of this class of protocols is difficult because they typically consist of complex behavioral characteristics, such as timed, probabilistic, and cryptographic operations.

To solve this problem, I proposed a probabilistic timed calculus for cryptographic protocols, called crypt\textsubscript{prob}\textsubscript{time}, and demonstrated how to use it for proving security or vulnerability of protocols. To the best of my knowledge, this is the first such process calculus that supports an expressive syntax and semantics, timed, probabilistic, and cryptographic issues at the same time. Hence, it can be used to verify systems that involve these three properties. For demonstration purposes, I applied crypt\textsubscript{prob}\textsubscript{time} to prove that both of the two previously proposed protocols, DTSN and SDTP, are vulnerable to the EAR flag setting attack, and the tricky sandwich attack. Taking into account the security holes in DTSN and SDTP, I proposed a new secured WSN transport protocol, called SDTP\textsuperscript{+}, and proved that the discussed attacks against DTSN and SDTP do not work in SDTP\textsuperscript{+}. I emphasize that although I only used crypt\textsubscript{prob}\textsubscript{time} to analyze WSN transport protocols, it can be applied to reasoning about other probabilistic, timed, cryptographic protocols as well.

Note that both the sr-calculus and the crypt\textsubscript{prob}\textsubscript{time} are based on the applied \(\pi\)-calculus, however, they are extended in different way to analyze different kind of protocols and systems. They are designed in different method, sr-calculus is entirely based on algebraic design with infinite process replication, while the crypt\textsubscript{prob}\textsubscript{time} is based on the probabilistic timed automaton design with recursive process invocation. It is an interesting question whether we can merge the two calculi into one complex calculi. I did not addressed this question so far, because my goal was to design methods that has as simple as syntax and semantics as possible. Merging the two calculi is not a straightforward task, because for each new syntax element we have to design carefully the matching formal semantics and bisimilarity definitions. For instance, we should consider how to define clock invariant, clock resets related to physical nodes and networks.

In addition, I proposed an automatic verification method, based on the PAT process analysis toolkit for this class of protocols, and used it to verify the security of the DTSN and SDTP protocols. To the best of my knowledge, PAT has not been used to verify WSN transport protocols before, however, I showed that it is well-suited for this purpose. Finally, my related papers in this topic are [Th13, 2013], [Th11, 2013], and [Th12, 2013].
Chapter 4

Query auditing for protecting sensitive information in statistical databases

4.1 Introduction

Query Auditing is a problem that has been studied intensively in the context of disclosure control in statistical databases [5]. The goal of an off-line query auditing algorithm is decide whether private information was disclosed by the responses of the database to a certain set of aggregate queries. Off-line query auditors work on queries received and responses provided in the past, therefore, they can only detect a privacy breach, but cannot prevent it. On-line query auditing algorithms, on the other hand, decide whether responding to a new incoming query would result in the disclosure of some private information, given the responses that have already been provided to past queries, and if responding to the new query would breach privacy, then the database can deny the response. Thus, on-line query auditing algorithms can prevent the unintended disclosure of private information.

To the best of my knowledge, in all existing works on query auditing, the private information whose disclosure one wants to detect or prevent consists of the sensitive fields of individual records in the database (e.g., the salary of a given employee). The reason may be that statistical databases are mainly used for computing statistics over certain attributes of human users (e.g., the average salary of women employees), and in such applications, each database record corresponds to an individual person. I define a novel setting for query auditing, where I want to detect or prevent the disclosure of aggregate values in the database (e.g., the maximum salary that occurs in the database), and I propose efficient off-line and on-line query auditing algorithms in this new setting.

The motivation behind my work comes from a project, called CHIRON (www.chiron-project.eu), where body mounted wireless sensor networks are used to collect medical data (e.g., ECG signals, blood pressure measurements, temperature samples, etc.) from a patient, and a personal device (e.g., a smart phone) is used to collect those data and provide controlled access to them for external parties (e.g., hospital personnel, personal coach services, and health insurance companies). In this context, the records stored in the database on the personal device all belong to the same patient, and individual values (i.e., sensor readings) may not be sensitive, whereas aggregates computed over those values (e.g., the maximum of the blood pressure in a given time interval) should be protected from unintended disclosure. The reason is that some of those aggregates (extreme values) can be used to infer the health status of the patient, and some of the accessing parties (e.g., health insurance companies) should be prevented to learn that information.

More specifically, in my dissertation, I study the problem of detecting or preventing the disclosure of the maximum (minimum) value in the database, when the querier is allowed to issue average queries to the database. I propose efficient off-line and on-line query auditors for this problem in the full disclosure model, and an efficient simulatable on-line query auditor in the par-
4. QUERY AUDITING FOR PROTECTING SENSITIVE INFORMATION IN STATISTICAL DATABASES

tional disclosure model. As for the organization of the dissertation, I start with an overview of the query auditing problem domain, introduce some terminology, review the state-of-the-art, and then present in detail the novel setting of query auditing problem and my proposed solutions for it.

4.2 Query Auditing Problems

Before going into details I introduce some notations for later use. Let \( n \) denotes the total number of records in the database. \( X = \{x_1, x_2, \ldots, x_n\} \) is the set of the private attribute values in the records. \( q = (Q, f) \) is an aggregate query, where \( Q \) specifies a subset of records, called the query set of \( q \). \( f \) is an aggregation function such as \( \text{MAX}, \text{MIN}, \text{SUM}, \text{AVG}, \text{MEDIAN} \). Finally, let \( a = f(Q) \) be the result of applying \( f \) to \( Q \), called the answer.

Query auditing problems can be classified according to Table 4.1. The auditor can be offline or online under various compromise models, namely, full disclosure and partial disclosure models. Partial disclosure can be further distinguished by probabilistic and interval disclosure. In addition, auditors can be simulatable which ensures provable privacy property. In the following, I briefly review each mentioned case.

In case of offline auditing, the auditor is given a set of \( t \) queries \( q_1, \ldots, q_t \) and the corresponding answers \( a_1, \ldots, a_t \), and its task is to determine offline if a breach of privacy has occurred. In contrast to offline auditors, an online auditor prevents privacy breach by denying to respond to a new query if doing so would lead to the disclosure of private information. More specifically, given a sequence of \( t - 1 \) queries \( q_1, \ldots, q_{t-1} \) that have already been posed and their corresponding answers \( a_1, \ldots, a_{t-1} \), when a new query \( q_t \) is received, the online auditor denies the answer if it detects that privacy can be breached, otherwise, it provides the (true) answer \( a_t \). The formal definition of auditors in the full disclosure model \[40\] is as follows:

**Definition 17.** An auditor is a function of the queries \( q_1, \ldots, q_t \) and the data set \( X \) that either gives an exact answer to the query \( q_t \) or denies the answer.

As for the compromise model, the privacy breach can be defined either based on a full disclosure or a partial disclosure model. In following subsections, I give an overview of each disclosure model, as well as the notion and concept of simulatable auditor.

4.2.1 Full Disclosure Model

In the full disclosure case, the privacy of some data \( x \) breaches when \( x \) has been uniquely determined. The formal definition of full disclosure model is as follows:

**Definition 18.** Given a set of private values \( X, X = \{x_1, x_2, \ldots, x_n\} \), a set of queries \( Q, Q = \{q_1, q_2, \ldots, q_t\} \), and the corresponding answers \( A, A = \{a_1, a_2, \ldots, a_t\} \). An element \( x_i \) is fully disclosed by \((Q, A)\) if it can be uniquely determined, that is, \( x_i \) is the same in all possible data sets \( X \) consistent with the answers \( A \) to the queries \( Q \).

As an illustrating example, let \( n = 3 \) and \( Q = \{\text{ALL, MAX}, \text{ALL, SUM}\} \). Assume that \( A = \{5, 15\} \) where \( \text{MAX}(x_1, x_2, x_3) = 5 \) and \( \text{SUM}(x_1, x_2, x_3) = 15 \), then one can deduce that \( x_1 = x_2 = x_3 = 5 \). Hence, \( x_i \) is fully disclosed for every \( i, i \in \{1, 2, 3\} \).

Based on this example, one may think that the full disclosure model is a weak definition since if a sensitive data can be deduced to lie in a very tiny interval, or in a large interval where the distribution is heavily skewed towards a particular value, then it is not considered a privacy breach. On the other hand, full disclosure model is strict in terms that there are situations where no query would ever be answered. Addressing these problems, researchers have proposed a definition of privacy that bounds the change in the ratio of the posteriori probability that a value \( x_i \) lies in an interval \( I \), given the queries and answers to the prior probability that \( x_i \in I \). This definition is known as probabilistic disclosure model \[51\], which I will introduce in the next subsection.
4.2. Query Auditing Problems

4.2.2 Partial/Probabilistic Disclosure Model

Consider an arbitrary data set $X = \{x_1, \ldots, x_n\}$, in which each $x_i$ is chosen independently according to the same distribution $\mathcal{H}$ on $(-\infty, \infty)$. Let $\mathcal{D} = \mathcal{H}^\circ$ denote the joint distribution.

I say that a sequence of queries and answers is $\lambda$-safe for an entry $x_i$ and an interval $I$ if the attacker’s confidence that $x_i \in I$ does not change significantly upon seeing the queries and answers.

**Definition 19.** The sequence of queries and answers, $q_1,\ldots, q_t$, $a_1,\ldots, a_t$ is said to be $\lambda$-safe with respect to a data entry $x_i$ and an interval $I \subseteq (-\infty, \infty)$ if the following Boolean predicate evaluates to 1:

$$Safe_{\lambda,i,I}(q_1,\ldots, q_t, a_1,\ldots, a_t) = \begin{cases} 1 & \text{if } 1/(1 + \lambda) \leq \frac{P_D(x_i \in I | a_{t-1} = f_t(q_t) = a_j)}{P_D(x_i \in I)} \leq (1 + \lambda) \\ 0 & \text{otherwise} \end{cases}$$

The definition below defines privacy in terms of a predicate that evaluates to 1 if and only if $q_1,\ldots, q_t$, $a_1,\ldots, a_t$ is $\lambda$-safe for all entries and all $\omega$-significant intervals: I say that an interval $J$ is $\omega$-significant if for every $i \in \{1,\ldots, n\}$, $P_D(x_i \in J)$ is at least $1/\omega$. I only care about the probability changes with respect to the so called significant intervals.

**Definition 20.** AllSafe$_{\lambda,\omega}(q_1,\ldots, q_t, a_1,\ldots, a_t) = \begin{cases} 1 & \text{if } Safe_{\lambda,i,J}(q_1,\ldots, q_t, a_1,\ldots, a_t) = 1, \forall J, i \in \{1,\ldots, n\} \\ 0 & \text{otherwise} \end{cases}$

For the probabilistic disclosure model, in the following I provide the definition of randomized auditor.

**Definition 21.** A randomized auditor is a randomized function of queries $q_1,\ldots, q_t$, the data set $X$, and the probability distribution $D$ that either gives an exact answer to the query $q_t$ or denies the answer.

Next I introduce the notion of $(\lambda, \omega, T)$-privacy game and $(\lambda, \delta, \omega, T)$-private auditor. The $(\lambda, \omega, T)$-privacy game between an attacker and an auditor, where in each round $t$ (for up to $T$ rounds):

1. The attacker (adaptively) poses a query $q_t = (Q_t, f_t)$.
2. The auditor decides whether to allow $q_t$ or not. The auditor replies with $a_t = f_t(Q_t)$ if $q_t$ is allowed, and denies otherwise.
3. The attacker wins if AllSafe$_{\lambda,\omega}(q_1,\ldots, q_t, a_1,\ldots, a_t) = 0$.

**Definition 22.** I say that an auditor is $(\lambda, \delta, \omega, T)$-private if for any attacker $A$

$$P[A \text{ wins the } (\lambda, \omega, T)\text{-privacy game}] \leq \delta.$$ 

The probability is taken over the randomness in the distribution $D$ and the coin tosses of the auditor and the attacker.

4.2.3 Online vs. Offline auditor

It is natural to ask the following question: Can an offline auditing algorithm directly solve the online auditing problem? More precisely, let $Q'$ be the subset of queries $q_1,\ldots, q_{t-1}$ that has been responded, and the corresponding answer set $A'$ of $a_1,\ldots, a_{t-1}$. When a new query $q_t$ is posed the offline auditor is activated with $(Q' \cup \{q_t\}, A' \cup \{a_t\})$. If some data is disclosed, then the answer is denied, otherwise, $a_t$ is returned.

Surprisingly, this method does not work in general, because even denials can leak information about the sensitive data. The next simple example illustrates the problem. Suppose that the
underlying data set is real-valued and that a query is denied only if some value is fully disclosed. Assume that the attacker poses the first query \( \text{SUM}(x_1, x_2, x_3) \) and the auditor answers 15. Suppose also that the attacker then poses the second query \( \text{MAX}(x_1, x_2, x_3) \) and the auditor denies the answer. The denial tells the attacker that if the true answer to the second query were given then some value could be uniquely determined. Note that \( \text{MAX}(x_1, x_2, x_3) \) should not be less than 5 since, otherwise, the sum could not be 15. Further, if \( \text{MAX}(x_1, x_2, x_3) > 5 \) then the query would not have been denied since no value could be uniquely determined. Consequently, \( \text{MAX}(x_1, x_2, x_3) \) must be equal to 5, and from this the attacker learns that \( x_1 = x_2 = x_3 = 5 \). One can deduce a crucial observation that query denials have the potential to leak information if in choosing to deny, the auditor uses information that is unavailable to the attacker (i.e., the answer to the current query). In order to overcome this problem, the concept of simulating auditor has been proposed by researchers.

4.2.4 Simulatable Auditing

Taking into account the crucial observation above, the main idea of simulating auditing is that the attacker is able to simulate or mimic the auditors decisions to answer or deny a query. As the attacker can equivalently determine for herself when her queries will be denied, she obtains no additional information about the sensitive data. For these reasons denials provably leak no information. The formal definition of simulatable auditor in full disclosure model is as follows:

**Definition 23.** An online auditor \( B \) is simulatable, if there exists another auditor \( B' \) that is a function of only \( Q \cup \{q_1\} = \{q_1, q_2, \ldots, q_t\} \) and \( A = \{a_1, a_2, \ldots, a_{t-1}\} \), and whose answer on \( q_t \) is always equal to that of \( B \).

When constructing a simulatable auditor for the probabilistic disclosure model, the auditor should ignore the real answer \( a_t \) and instead make guesses about the value of \( a_t \), say \( a'_t \), computed on randomly sampled data sets according to the distribution \( D \) conditioned on the first \( t-1 \) queries and answers. The definition of simulatable auditor in the probabilistic case is given in Definition 24.

**Definition 24.** Let \( Q_t = (q_1, \ldots, q_t), A_{t-1} = (a_1, \ldots, a_{t-1}) \). A randomized auditor \( B \) is simulatable if there exists another auditor \( B' \) that is a probabilistic function of \( (Q_t, A_{t-1}, D) \), and the outcome of \( B \) on \( (Q_t, A_{t-1} \cup \{a_t\}, D) \) and \( X \) is computationally indistinguishable from that of \( B' \) on \( (Q_t, A_{t-1}, D) \).

A general approach for constructing simulatable auditors: The general approach, shown in Fig. 4.1, works as follows: The input of the auditor is the past \( t-1 \) queries along with their corresponding answers, and the current query \( q_t \). As mentioned before, the auditor should not consider the true answer \( a_t \) when making a decision. Instead, to make it simulatable for the attacker, the auditor repeatedly selects a data set \( X \) consistent with the past \( t-1 \) queries and answers, and computes the answer \( a'_t \) based on \( q_t \) and \( X' \). Then, the auditor checks if answering with \( a'_t \) leads to a privacy breach. If a privacy breach occurs for any consistent data set (full disclosure model) or for a large fraction of consistent data sets (partial disclosure model), the response to \( q_t \) is denied. Otherwise, the true answer \( a_t \) for \( q_t \) is returned.

While ensuring no information leakage, a simulatable auditor has the main drawback that it can be too strict, and deny too many queries resulting in bad utility. In the full disclosure model, if any of the possible answers \( a'_t \) could lead to the disclosure of sensitive data, then the simulatable auditor will deny every query. To show this, let us revisit the example above, and let \( n = 3 \) and \( X = \{x_1, x_2, x_3\} \), and their value is \( x_1 = 3, x_2 = 7, x_3 = 5 \). The goal is to prevent the full disclosure of \( x_1 \). For the first query \( q_1 = \text{SUM}(x_1, x_2, x_3) \) the answer is returned, \( a_1 = 15 \). However, the second query \( q_2 = \text{MAX}(x_1, x_2, x_3) \) will always be denied by a simulatable auditor, even if for \( x_1 = 3, x_2 = 7, x_3 = 5 \), it is safe to respond. This is because simulatable auditor does not consider the true value of \( x_1, x_2, x_3 \) when making a decision. The auditor found that there is a data set consistent with \( (q_1, a_1) \) that would lead to the full disclosure of \( x_1 \), namely, \( x_1 = x_2 = x_3 = 5 \), therefore it always deny the second query.
4.3. Related Works

In this section, I discuss in details the state-of-the-art about the works on query auditing problem, which is illustrated in Table 4.1. I discuss the related works according to the online and offline problems, as well as the compromise models and the simulatability. I note that all the related works discussed below are concerned with protecting the privacy of individual values, and not aggregated values that I am addressing.

In case of full disclosure model, efficient simulatable online auditors have been proposed for SUM [21], MAX, MIN and the combination of MAX and MIN queries [40], [51]. In all these cases the values of private attributes are assumed to be unbounded real numbers. For effectiveness, the MAX and MIN auditors assume that there is no duplication among $x_1, \ldots, x_n$ values. An interesting future work could be proposing methods for solving these limiting assumptions.

Effective offline auditors have been proposed for MAX, MIN, MAX and MIN, SUM queries over unbounded real values and under the same conditions as in the online case [21], [22]. In addition to these, SUM auditors have also been proposed for boolean values [42]. The authors proved that the offline sum auditing problem over boolean values is coNP-hard. Then they gave an efficient offline sum auditing algorithm in the case that the queries are “one-dimensional”, i.e., for some ordering of the elements, say $x_1, \ldots, x_n$, each query involves a consecutive sequence of values $x_i, x_{i+1}, x_{i+2}, \ldots, x_j$. Finally, it has been shown that the problem of auditing the combination of MAX and SUM (MIN and SUM, and the combination of MIN and MAX and SUM) queries are

---

**Table 4.1:** Summary of query auditing problems and related works. The abbreviation Sim. means Simulatable.

<table>
<thead>
<tr>
<th>Prob. Disc.</th>
<th>Online Auditing</th>
<th>Offline Auditing</th>
<th>Handl. Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Disc.</td>
<td>Sim. MAX, MIN, MAX &amp; MIN - real, unbound values</td>
<td>MAX &amp; SUM (NP-hard) - real, unbound values</td>
<td>MAX, MIN, MAX &amp; MIN - delete, modify, insert</td>
</tr>
<tr>
<td></td>
<td>Sim. SUM - real, unbound values</td>
<td>SUM - real, unbound values</td>
<td>SUM - delete, modify, insert</td>
</tr>
<tr>
<td></td>
<td>Sim. MAX, MIN, MAX &amp; MIN - real, unbound values</td>
<td>MAX, MIN, MAX &amp; MIN - real, unbound values</td>
<td></td>
</tr>
</tbody>
</table>
4. QUERY AUDITING FOR PROTECTING SENSITIVE INFORMATION IN STATISTICAL DATABASES

NP-hard [21].

In [73] an offline SUM auditor has been proposed in which sensitive information about individuals is said to be compromised if an accurate enough interval is obtained into which the value of the sensitive information must fall. This work corresponds to the Interval disclosure part of Table 4.1. In [20] the authors consider the problem of auditing queries where the result is a distance metric between the query input and some secret data.

Similarly, simulatable SUM, MAX, MIN and the combination of MAX and MIN auditors have been proposed for probabilistic disclosure model [21], [22]. In all cases the private attributes are assumed to take their values randomly according to uniform and log-concave distributions. In [45] the notion of simulatable binding has been proposed that provides better utility than simulatable auditor, but requires more computations.

In many works in the past, when designing an auditor, it is assumed that the database is fix. Unfortunately, this is often not true in the real-life databases. Hence, targetting this issue, in the full disclosure model, auditors have been proposed that taking into account the possibility of deleting, modifying, inserting records in the database. Such auditors have been proposed for MIN, MAX, MIN and MAX, and SUM queries [52]. In the following, I review a bit more in details the offline SUM auditor proposed in [21] because it will be referred to during discussing my method.

Offline SUM auditor: In [21] the authors proposed a method for auditing SUM queries over unbounded real values. In this method, each query is expressed as a row in a matrix with a 1 wherever there is an index in the query and a 0 otherwise. If the matrix can be reduced to a form where there is a row with one 1 and the rest 0s then some value has been compromised. Such a transformation of the original matrix can be performed via elementary row and column operations. This auditor is simulatable because the answers to the queries are ignored when the matrix is transformed. In the rest of the dissertation I will refer to this algorithm as $A_{\text{sum}}$.

4.4 My contributions

My main contributions are the following: I address a new auditing problem by considering an aggregation value of a dataset to be sensitive and concentrating on protecting the privacy of aggregation values. I propose both offline and online auditors in the full disclosure model, as well as simulatable auditor in the partial disclosure model. All the three auditors are constructed based on polynomial time algorithms.

My proposed offline and online auditors in the full disclosure model are novel compared with the related works. Furthermore, I note that in case of simulatable auditor in the partial disclosure model, the methods proposed for SUM auditors [40], [39] cannot be used entirely in my case, and although some parts of my Lemmas use similar parameters, the proofs are not the same. For instance, in the proof of Lemma 12, because the domain of each $x_i$ is bounded, I have to take into account additional cases. Moreover, in Algorithm 4, I had to use different parameters from that in the related works to solve my problem, and the proof of Theorem 3 is also different. Finally, I propose the generalized parameters in Lemma 13 instead of specific parameters. Due to page limitation, the proof of the Lemmas 12, 13, 14, Theorem 3 and the general parameter choice can be found in the Appendices A – D of my long report [Th10, 2012].

In contrast to the previous works, I assume that the domain of sensitive values is bounded, which leads to some new problems. Note that in each case below, without loss of generality and for simplicity, I transform each equation $\sum_{k}^{x} x_i = a$ induced by each AVG query and its answer to the form $\sum_{k}^{x} x_i = ak$.

As for the attacker model, I assume that there is only one attacker at a time, hence, I do not deal with the collusion attackers case. Moreover, I consider only one session at a time, not interleaving sessions. Moreover, within a session the attacker repeatedly poses average queries and its goal is to deduce somehow the maximum or minimum values. The attacker can use any algorithm to compute the secrets based on the queries and answers.

In the rest of the dissertation, I denote the auditor that gets average queries and protect the privacy of the max (min) value as Auditor$_{\text{avg}}^{\text{max}}$ (Auditor$_{\text{avg}}^{\text{min}}$). Moreover, I denote the maximum of
the $n$ values $x_1, \ldots, x_n$, by MAX. Finally, in the dissertation I mainly focus on the privacy of the maximum value, however, the auditors can be constructed for minimum value in the same way. I briefly discuss about the MIN case in [Th10 , 2012].

4.5 Offline Auditor$^{\text{max}}_{\text{avg}}$ in the full disclosure model

The problem is defined as follows: Given $t$ queries $q_1, \ldots, q_t$ over the stored data set $X = \{x_1, \ldots, x_n\}$ and their corresponding answers $a_1, \ldots, a_t$. Each query $q_i$ is of the form $(Q_i, \text{AVG})$, where $Q_i \subseteq [n]$, and the value of each $x_i$ is assumed to be a real number that lies in a finite interval $[\alpha, \beta]$, where $\beta > \alpha$. The task of the offline auditor is to detect if the value of MAX is fully disclosed.

The AVG queries and answers can easily be transformed to the form that is equivalent to the SUM queries and answers case by multiplying each query and answer with the denominators. Hence, one trivial question is that can we apply directly the algorithm $A_{\text{sum}}$ to this problem? The answer is negative because except for the case when all the $x_i$-s, $i \in \{1, \ldots, n\}$, can be uniquely determined, $A_{\text{sum}}$ cannot tell surely anything about the exact value of MAX. This is because $A_{\text{sum}}$ does not take into account the answers and the bounds $\alpha, \beta$ of each $x_i$. For the purpose of illustration, let us take the following example: let $X = \{x_1, x_2, x_3\}$ and $\forall x_i \in [20, 90]$, let $q_1 = (\{x_1, x_2\}, \text{AVG})$, $q_2 = (\{x_1, x_2, x_3\}, \text{AVG})$ and the corresponding answers $a_1 = 45$, $a_2 = 60$. Finally, let the stored values be $x_1 = 40$, $x_2 = 50$, $x_3 = 90$. According to $A_{\text{sum}}$ the value of MAX is not fully disclosed, because the answers and the bounds of $x_i$’s are not considered. We only know that $x_3$ can be uniquely determined, but nothing about its value. However, in fact MAX is fully disclosed because by involving the answers we additionally know that the value of $x_3$ is 90, which at the same time is the value of MAX since 90 is the upperbound of any $x_i$. Hence, I have to consider a method that also takes into account the bounds of $x_i$’s and the results. For this purpose, I propose the application of the well-known linear optimization problem.

The linear optimization problem for the offline auditor can be defined as follows: The $t$ queries can be represented by a matrix $A$ of $t$ rows and $n$ columns. The corresponding answers are represented by a column vector $b$.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{t,1} & a_{t,2} & \cdots & a_{t,n} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_t \end{pmatrix}$$

Each row $r_i = (a_{i,1}, \ldots, a_{i,n})$ of $A$ represents the query set $Q_i$ of the query $q_i$. The value of $a_{i,j}$, $1 \leq i, j \leq n$, is 1 wherever $x_j$ is in the query set $Q_i$, and is 0 otherwise. However, I note that my method also works in such application where queries require $a_{i,j}$ to be any real value. Each element $b_i$ of $b$ represents the answer for the query $q_i$. Since each attribute $x_i$ takes a real value from a bounded interval $[\alpha, \beta]$ we obtain the following special linear equation system, also known as feasible set, which includes equations and inequalities:

$$\mathcal{L} = \{ \tilde{A}\tilde{x} = \tilde{b}, \text{where } \tilde{x} \text{ is the vector } (x_1, \ldots, x_n)^T. \}
\quad \alpha \leq x_i \leq \beta, \forall x_i : x_i \in \{x_1, \ldots, x_n\}$$

For instance, let $x_i \in [0, 5]$ for every $x_i$ (1 $\leq i \leq 4$), and let the first query and answer be $AVG(x_1, x_2) = 3$. In addition, let the second query and answer be $AVG(x_3, x_4) = 5$. The corresponding feasible set of this example, denoted by $\mathcal{L}'$, is as follows:

$$\mathcal{L}' = \{ \tilde{A}\tilde{x} = \tilde{b}, \text{where } \tilde{x} \text{ is the vector } (x_1, x_2, x_3, x_4)^T. \}
\quad 0 \leq x_i \leq 5, \forall x_i : x_i \in \{x_1, x_2, x_3, x_4\}$$

where
4. QUERY AUDITING FOR PROTECTING SENSITIVE INFORMATION IN STATISTICAL DATABASES

\[
\tilde{A} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}, \quad \tilde{b} = \begin{pmatrix} 6 \\ 10 \end{pmatrix}
\]

Then, by appending each objective function maximize(\(x_i\)) to \(L\), we get \(n\) linear programming problems \(P_i\), for \(i \in \{1, \ldots, n\}\). Let \(x^{\max}_i = \text{maximize}(x_i)\), then the maximum value of \(x_1, \ldots, x_n\) is the maximum of the \(n\) maximized values, \(x^{\text{opt}} = \max\{x^{\max}_1, \ldots, x^{\max}_n\}\). Let us denote the whole linear programming problem above for determining the maximum value \(x^{\text{opt}}\) by \(P\). Note that \(x^{\text{opt}}\) returned by \(P\) is the exact maximum value (i.e., equal to the stored maximum) if \(i\) \(L\) has a unique solution or \((ii)\) \(L\) does not have a unique solution but based on \(L\) there exist some \(x_i\) that can be derived to be equal to \(x^{\text{opt}}\). To see the meaning of point \((ii)\), let us consider the specific case of \(L\) in which \(n = 4\), \(\alpha = 0\), \(\beta = 5\), and \(\tilde{A} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}, \tilde{b} = \begin{pmatrix} 6 \\ 10 \end{pmatrix}\). In this example, \(L\) does not have a unique solution but the exact maximum still can be derived such that \(x_3 = x_4 = 5\).

Otherwise, \(x^{\text{opt}}\) is the best estimation of the exact maximum. Note that in my case \(L\) always has a solution, because one possible solution is actually the values stored in the database.

Based on this linear programming problem, my offline auditor will follow the next steps. Given \(t\) queries \(q_1, \ldots, q_t\) over \(X = \{x_1, \ldots, x_n\}\) and their corresponding answers \(a_1, \ldots, a_t\), the value of \(\text{MAX}\) is fully disclosed in any of the following two cases:

- \((F1)\) In case \(L\) has a unique solution, the value of \(\text{MAX}\) is equal to \(x^{\text{opt}}\).
- \((F2)\) In case \(L\) does not have a unique solution: If by following the solving procedure of \(L\) (e.g., basic row and column operations), there exist some \(x_i\) that can be uniquely determined such that \(x_i = x^{\text{opt}}\), then the value of \(\text{MAX}\) is \(x_i\). This is because \(x^{\text{opt}}\) is always at least as large as the value of \(\text{MAX}\).

Otherwise, the attacker cannot uniquely deduce the value of \(\text{MAX}\). The case \((F1)\) is straightforward, the attacker computes the maximum of the \(n\) derived values. Case \((F2)\) is similarly simple: Since the \(x^{\text{opt}}\) returned by \(P\) is always at least as large as the real maximum, if any \(x_i\) is equal to \(x^{\text{opt}}\) then it must be the real, stored maximum. Otherwise, because the values of all the \(x_i\)'s are real numbers in the interval \([\alpha, \beta]\) where \(\beta > \alpha\), the number of the consistent values of the data set is infinite.

The complexity of the offline auditor: The complexity of the offline auditor is based on the complexity of \(P\). It is well-known that there are polynomial time linear programming methods to solve \(P\), for instance, the class of interior point based methods like the ellipsoid algorithm [12] with the complexity of \(O(n^3L)\), the projective algorithm [38] with \(O(n^{1.3}L)\), and the path-following algorithm [60] with \(O(n^4L)\). Here \(n\) is the number of variables while \(L\) is the size of the input in bits, and the number of rows is assumed to be \(O(n)\). However in practice, I believe that, like in many papers, (e.g., [53]), the exponential worst-case complexity simplex method [12] is the most effective to solve my problem. To summarize, my offline auditing method has a polynomial time complexity in the worst case.

4.6 Online Auditor\(^{\text{max}}\) in the full disclosure model

The online auditing problem is defined as follows: Given \(t - 1\) queries \(q_1, \ldots, q_{t-1}\) of form \((Q_t, \text{AVG})\) over the stored data set \(X = \{x_1, \ldots, x_n\}\) and the corresponding answers \(a_1, \ldots, a_{t-1}\). The value of each \(x_i\) is assumed to be real number that lies in a finite interval \([\alpha, \beta]\), \(\beta > \alpha\). When a new \(q_t\) is posed, the task of the online auditor is to make a decision in real-time whether to answer or deny the query. More specifically, my goal is to propose an auditor that detects if answering with true \(a_t\) causes full disclosure of \(\text{MAX}\). First of all, I discuss the construction of a simulatable auditor for this problem, and I will show the limitation of simulatable auditor in this case. Thereafter, I propose another method that gets around this limitation.

Note that in Algorithm 1, based on the concept of simulatable auditor shown in Fig. 4.1, by ignoring the true answer \(a_t\) we examine every data set \(X'\), consistent with the past queries.
4.6. Online Auditor$_{\text{avg}}^{\text{max}}$ in the full disclosure model

and answers, and check if it causes the full disclosure of MAX. This means that the answer $a'_i$ computed based on $X'$ and $Q_i$, is included in the analysis. The auditor is simulatable because it never looks at the true answer when making a decision. The main drawback, however, of using simulatable auditor in my problem is the bad utility. In order to see this, consider any AVG query $q$ that specifies a subset $\{x_{i1}, x_{i2}, \ldots, x_{ik}\}$ of $X$ as the query set. There always exist a data set $X'$ for which this query is not safe to respond, namely, the data set where $x_{i1} = x_{i2} = \ldots = x_{ik} = \beta$, as in this case, the true response would be $\beta$, and the queryer can figure out that all values in the query set must be equal to $\beta$. This essentially means that all queries should be denied by a simulatable auditor.

**Algorithm 1**: Simulatable online auditor Auditor$_{\text{avg}}^{\text{max}}$

Inputs: $q_1, \ldots, q_t, a_1, \ldots, a_{t-1}, \alpha, \beta$;

for each consistent data set $X'$ do compute the AVG $a'_i$ based on $Q_t$ and $X'$;

Let $L_i$ be the feasible set formed by the $t$ queries/answers;

if $L_i$ yields an exact maximum then output DENY; endif

endfor

output $a_t$;

**Algorithm 2/a**: Online Auditor Auditor$_{\text{avg}}^{\text{max}}$

Inputs: $q_1, \ldots, q_t, a_1, \ldots, a_{t-1}, d_t, \alpha, \beta$;

Let $L'_i$ be the feasible set formed by the $t$ queries/answers;

Let $x_{\text{avg}}^\ast$ be the returned maximum by solving $P$ with $L'_t$

if $|x_{\text{avg}}^\ast - \text{MAX}| > d_t$, AND $(\text{MAX} - \text{MAX}) > d_t$ then output $a_t$; endif

else if $|x_{\text{avg}}^\ast - \text{MAX}| \leq d_t$, OR $(\text{MAX} - \text{MAX}) \leq d_t$, then output DENY; endif

To achieve better utility, hence, I propose two methods (Algorithms 2/a and 2/b) that are not simulatable but I show that they still ensure, in the full disclosure model, the privacy of the maximum value. I start with discussing the Algorithm 2/a: Let us denote $|x_{\text{avg}}^\ast - \text{MAX}|$ as the absolute distance between $x_{\text{avg}}^\ast$ and MAX. Let $\text{MAX}_i$ be the maximum of the first $t$ answers. Let $L^\ast$ be the feasible set that is similar to $L$ but the constraint $\alpha \leq x_i \leq \beta$ is involved only for such $x_i$’s that occurs in the first $t$ queries, and not for all the $n$ variables. Namely, in $L^\ast$ the second line of $L$ is changed to $\alpha \leq x_i \leq \beta$, for all $i$ such that $x_i$ occurs in the first $t$ queries.

$$L^\ast = \left\{ \begin{array}{l} \bar{A}\bar{x} = \bar{b}, \text{where } \bar{x} \text{ is the vector } (x_1, \ldots, x_n)^T. \\ \alpha \leq x_i \leq \beta, \forall x_i \text{ that occurs in the first } t \text{ queries.} \end{array} \right.$$  

Note that I use $L^\ast$ instead of my online auditor because by doing this the auditor leaks less information to the attacker either when answering or denying. To illustrate this, let us consider the example in which the data set is $\{x_1, x_2, x_3, x_4\}$ and $\forall x_i : x_i \in [0, 5]$. Assume also that the first query $q_1$ is $2x_1 + x_2$, and its corresponding answer is 4. The feasible set $L$ induced by these pieces of information is as follows:

$$L = \left\{ \begin{array}{l} \bar{A}\bar{x} = \bar{b}, \text{where } \bar{x} \text{ is the vector } (x_1, x_2, x_3, x_4)^T. \\ 0 \leq x_i \leq 5, \forall x_i : x_i \in \{x_1, x_2, x_3, x_4\} \end{array} \right.$$  

where

$$\bar{A} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ \end{pmatrix}, \quad \bar{b} = \begin{pmatrix} 8 \end{pmatrix}$$  

However, in this situation the attacker knows that the value of $x_{\text{avg}}^\ast$ returned by $P$ is always 5 because during estimating the maximum, all the possible values of the four variables are considered and involved. In contrast, by using $L^\ast$ in the previous example, the value of $x_{\text{avg}}^\ast$ returned by $P$ is the maximum of only the variables $x_1$, $x_2$ that occur in $q_1$. This means that the value of $x_{\text{avg}}^\ast$ depends on the true answer of $q_1$, hence, the auditor does not know exactly the value of $x_{\text{avg}}^\ast$ without getting the true answer.

The online auditor, based on the Algorithms 2/a, works as follows: Recall that $L^\ast$ is defined over $t$ queries and answers. Whenever a new query $q_t$ is posed, the auditor computes the true
answer $a_i$, and then it solves the problem $P$ with $\mathcal{L}^*$, obtaining $x^{opt}$. If for a given threshold value $d_{tr}, |x^{opt} - \text{MAX}| > d_{tr}$ and $(\text{MAX} - \text{max}_{\alpha}) > d_{tr}$ then the true answer $a_i$ is provided. Otherwise, if $|x^{opt} - \text{MAX}| \leq d_{tr}$ or $(\text{MAX} - \text{max}_{\alpha}) \leq d_{tr}$ the auditor denies.

In the following, I continue with discussing the Algorithm 2/b, which provides better utility than the Algorithm 2/a: I note that using directly algorithm $A_{\text{sum}}$ or the proposed offline auditor in Section 4.5 for constructing an online auditor does not work because denying could lead to full disclosure of MAX. For instance, according to the offline auditor, if the feasible set $\mathcal{L}$ is such that there is a variable $x_i$ that can be uniquely determined to be the upperbound, the auditor should deny. However, by receiving a deny the attacker knows that the particular variable is the upperbound since otherwise the auditor should have answered.

Nevertheless, with a minor modification, I still can include the concept of the offline auditor in Section 4.5. Namely, the auditor denies to answer not when there is a variable $x_i$ that can be uniquely inferred to be the upperbound, but instead, it denies when a variable is deduced to be within the tolerance threshold $d_{tr}$ from the stored maximum. The proposed online auditor based on the concept of offline auditor is as follow:

**Algorithm 2/b:** Online auditor $A_{\text{sum}}^{\text{MAX}}$

**Inputs:** $q_1, \ldots, q_t, a_1, \ldots, a_t, d_{tr}, \alpha, \beta$.

Let $\mathcal{L}_t$ be the feasible set formed by the $t$ queries/answers

if with $\mathcal{L}_t$ the linear equation system has unique solution then output DENY; return; endif

else if there is a $x_i$ that can be uniquely determined then

if $(\text{MAX} - x_i) > d_{tr}$ AND $(\text{MAX} - \text{max}_{\alpha}) > d_{tr}$ then output $a_i$; return; endif

else if $(\text{MAX} - x_i) \leq d_{tr}$ OR $(\text{MAX} - \text{max}_{\alpha}) \leq d_{tr}$ then output DENY; return; endif

endif else output $a_i$;

This online auditor prevent MAX from being fully disclosed, because in case of deny, the attacker still cannot gain any information about the exact value of MAX, at most, it only knows that the value is outside the tolerance threshold $d_{tr}$ from MAX. This version of online auditor produces better utility than Algorithm 2/a, because it does not depend on the uncertainty of the estimation of MAX. The auditor based on Algorithm 2/a can deny a huge number of queries in case the stored maximum MAX is within the tolerance threshold $d_{tr}$ from the upperbound $\beta$. For instance, let consider the example data set $x_1 = 3, x_2 = 4, x_3 = 1$, and the bounds of the variables $\alpha = 0, \beta = 5$. In case the treshold $d_{tr}$ is 2, the auditor based on Algorithm 2/a will deny the queries $q_1 = \text{AVG}(x_3)$ because the estimated maximum based on $\mathcal{L}$ formed by the first queries, will be the upperbound 5, which is within the treshold from the stored maximum 4. This scenario cannot happen in Algorithm 2/b.

**Lemma 6.** Assuming that $d_{tr} > 0$, the online auditor implemented by the Algorithms 2/a and 2/b provides the privacy of MAX in the full disclosure model.

**Proof.** (Sketch taken from the section VI.B of my report [Th10, 2012]) Let $f_{\text{att}}(d_{tr}, q_1, \ldots, q_t, a_1, \ldots, a_{t-1}, \alpha, \beta)$ represent the attacker’s based on the input parameters, and returning as output a deny or an answer. I prove that my online auditors do not leak information about MAX, in the full disclosure model by showing that the number of the data sets and the parameter sets for which $f_{\text{att}}$ returns deny or answer is always larger than 1. In other words, in every possible scenario, for the attacker the number of possible maximum values will always be greater than 1, hence, the value of MAX cannot be uniquely determined. I apply mathematical induction in each case to show this.

The complexity and utility of the online auditors: The worst-case complexity of the online auditor depends on the worst-case complexity of $P$ and the number of posed queries. We can assume that the number of queries is $O(n)$, where $n$ is the size of the data set. In this case, by applying one of the polynomial time linear program solver methods, the whole complexity remains polynomial.
The utility of the auditor can be measured based on the number of denies. This is controlled by the threshold value \(d_{tr}\). Broadly speaking, if \(d_{tr}\) is large then the expected number of denies is greater, while when \(d_{tr}\) is small the degree of privacy provided decreases, because the estimated maximum can be very close to the real maximum \(\text{MAX}\). The more specific choice of \(d_{tr}\) to achieve a good trade-off between utility and privacy level for the specific application scenarios is an interesting question, for which I will find the answer in the future work.

### 4.7 Simulatable auditor\(^\text{max}_{\text{avg}}\) in the probabilistic disclosure model

The auditing problem under probabilistic disclosure model is defined as follows: Given \(t-1\) queries \(q_1, \ldots, q_{t-1}\) of form \((Q_i, \text{AVG})\) over the stored data set \(X = \{x_1, \ldots, x_n\}\) and the corresponding answers \(a_1, \ldots, a_{t-1}\), a new \(q_t\) is posed, and the task of the auditor is to make a decision in real-time whether to answer or deny the query. My goal is to propose a simulatable auditor that prevents a probabilistic disclosure of \(\text{MAX}\).

By transforming the AVG queries to SUM queries I can adapt one part of the auditor given in [40], [39]. However, I note that my problem is different from those in [40], [39], because I consider bounded intervals and MAX instead of unbounded domain and individual values. Hence, the methods proposed for SUM auditors cannot be used entirely in my case, and although some parts of my Lemmas use similar parameters, the proofs are not the same (see the Appendices). For instance, in the proof of Lemma 12, because the domain of each \(x_i\) is bounded, I have to take into account additional cases. Moreover, in Algorithm 4 I had to use different parameters from that in the related works to solve my problem, and the proof of Theorem 3 is also different. Finally, I propose the general parameter choice in Lemma 13 instead of specific parameters.

I assume that each element \(x_i\) is independently drawn according to a distribution \(\mathcal{G}\) that belongs to the family of log-concave distributions over the set \(\mathbb{R}\) of real numbers in \([\alpha, \beta]\). Note that I consider the class of log-concave distribution because it covers many important distributions including the gaussian distribution. In addition, my online simulatable auditor is based on random sampling and I want to apply directly the method of Lovasz [46] on effective sampling from log-concave distributions. The main advantage of the sampling method in [46] is that it is polynomial-time and produces only small error.

Next, I give the formal definition of the log-concave distribution: A distribution over a domain \(D\) is said to be log-concave if it has a density function \(f\) such that the logarithm of \(f\) is concave. Specifically, the density function \(f : D \mapsto \mathbb{R}_+\) is log-concave if it satisfies \(f(tx + (1-t)y) \geq f(x)^t f(y)^{1-t}\) for every \(x, y \in D\) and \(0 \leq t \leq 1\).

I give some relevant points that will make the method in [46] applicable in the construction of my auditor. Let us denote the distribution of \(\text{MAX}\) as \(\mathcal{G}_{\text{max}}\). The attacker’s initial knowledge about \(\text{MAX}\) can be given as the apriori probability \(P_{\mathcal{G}_{\text{max}}} (\text{MAX} \in I)\) for some \(I \subseteq [\alpha, \beta]\).

**Lemma 7.** The truncated version of log-concave distribution is also log-concave.

**Proof.** Let the density and the cumulative distribution function of a variable \(Y\) be \(f(y)\) and \(F(y)\), respectively. The truncated version of \(f(y)\), \(f(y | Y \in I)\), is equal to \(\frac{f(y)}{\int_{y \in I} f(y) dy}\). By assumption, \(f(y)\) is log-concave and the denominator is a constant, it follows that \(f(y | Y \in I)\) is log-concave. Hence, returning to my problem, each \(x_i\) is taken according to a truncated log-concave distribution, which is log-concave.

**Lemma 8.** The product of log-concave distributions is also log-concave.

**Proof.** This comes from the fact that the logarithm of the product of log-concave functions is a concave function.

**Lemma 9.** If \(\mathcal{G}\) is log-concave distribution then the joint distribution of \((x_1, \ldots, x_n)\), \(\mathcal{G}^n\), is also log-concave.
4. QUERY AUDITING FOR PROTECTING SENSITIVE INFORMATION IN STATISTICAL DATABASES

Proof. This comes immediately from Lemma 8.

Lemma 10. If the distribution $\mathcal{G}$ is log-concave then $\mathcal{G}_{\text{max}}$ is log-concave.

Proof. Because $x_1, \ldots, x_n$ are drawn independently from the same distribution, the cumulative distribution function $F_{\mathcal{G}_{\text{max}}}(x)$ is $[F_{\mathcal{G}}(x)]^n$. Hence, by executing derivation we get that the density function $f_{\mathcal{G}_{\text{max}}}(x)$ is equal to $n[F_{\mathcal{G}}(x)]^{n-1}f_{\mathcal{G}}(x)$. It is well-known that the cumulative distribution function of a log-concave distribution is log-concave [10]. Hence, because $n$ is positive, by applying Lemma 8 the result follows immediately.

Lemma 11. Let $\mathcal{G}_t^n$ be the joint distribution $\mathcal{G}^n$ conditioned on the first $t$ queries and answers, $\wedge_{j=1}^t(\text{avg}(Q_j) = a_j)$. Then, if $\mathcal{G}$ is a log-concave distribution then $\mathcal{G}_t^n$ is also log-concave.

Proof. Similar to the truncated distribution density function, the density of $\mathcal{G}_t^n$ is as follows: $f_{\mathcal{G}_t^n}(\cdot) = f_{\mathcal{G}_t^n}(\cdot)\frac{f_{\mathcal{G}_t^n}(\cdot)}{f_{\mathcal{G}_t^n}(\cdot)}$, where $f_{\mathcal{G}_t^n}(\cdot)$ is the density of the joint distribution, $I_p(\cdot)$ is an indicator function that returns 1 if $\bar{x}$ are in the convex constraint $\mathcal{P}$ induced by the $t$ queries and answers, and 0 otherwise. The denominator contains the probability that $\bar{x}$ being within $\mathcal{P}$, which is a constant value for a given $\mathcal{P}$. According to Lemma 8 and based on the similar argument as in the case of Lemma 7, it follows that $f_{\mathcal{G}_t^n}(\cdot)$ is log-concave.

In my case, the predicate $\lambda$-Safe and AllSafe is a bit differ from the traditional definitions discussed in Section 4.2.2, since I am considering the maximum of $n$ values instead of single values. Hence, the definitions are modified as follows:

Definition 25. The sequence of queries and answers, $q_1, \ldots, q_t, a_1, \ldots, a_t$ is said to be $\lambda$- Safe with respect to an interval $I \subseteq [\alpha, \beta]$ if the following Boolean predicate evaluates to 1:

$$\text{Safe}_{\lambda,I}(q_1, \ldots, q_t, a_1, \ldots, a_t) =$$

$$\begin{cases} 1 & \text{if } 1/(1 + \lambda) \leq \frac{P_{\mathcal{G}_{\text{post}}}^{\text{MAX} \in I}(|\wedge_{j=1}^t(\text{avg}(Q_j) = a_j))}{P_{\mathcal{G}_{\text{max}}}^{\text{MAX} \in I}} \leq (1 + \lambda) \\ 0 & \text{otherwise} \end{cases}$$

where $\mathcal{G}_{\text{post}}$ is the distribution of the posteriori probability, and $\mathcal{G}_{\text{max}}$ is the distribution of MAX. The definition of AllSafe is then given over all $\omega$-significant intervals $J$ of $[\alpha, \beta]$. Here the notion of $\omega$-significant interval is defined over the maximum value instead of individual values as in Section 4.2.2: An interval $J$ is $\omega$-significant if $P_{\mathcal{G}_{\text{max}}}(\text{MAX} \in J) \geq \frac{1}{2}$.

Definition 26. AllSafe$_{\lambda,\omega}(q_1, \ldots, q_t, a_1, \ldots, a_t) =$

$$\begin{cases} 1 & \text{if Safe}_{\lambda,I}(q_1, \ldots, q_t, a_1, \ldots, a_t) = 1, \forall J \\ 0 & \text{otherwise} \end{cases}$$

Note that in each definition, compared to the Definitions 19 and 20, the dependence on the parameter $i$ has been removed. The definition of the $(\lambda, \omega, T)$-privacy game and the $(\lambda, \delta, \omega, T)$-privacy auditor remains unchanged.

In [46] the authors proposed the algorithm Sample($D, \epsilon$) for sampling from an arbitrary log-concave distribution $D$ (defined in $\mathbb{R}^n$) with the best running time of $O^*(n^2)$, such that the sampled output follows a distribution $D'$ where the total variation distance between $D$ and $D'$ is at most $\epsilon$. The notation $O^*$() is taken from [46], and indicates that this best running time does not show the polynomial dependence on log $n$, and the error parameter $\epsilon$. I make use of this algorithm for constructing my auditor.

The next question is that what kind of, and how many intervals $I$ we need to consider when examining the AllSafe predicate. Of course, in practise, we cannot examine infinite sub-intervals in $[\alpha, \beta]$. Following the approach in [40], I show that it is enough to check only finite number of intervals.

Let us consider the well-known notion of quantiles and quantile function in statistics. Broadly speaking, quantiles are points taken at regular intervals from the cumulative distribution function...
of a random variable, and divide ordered data into essentially equal-sized data subsets. In other words, a \( p \)-quantile is such a value of \( x \) for which the fraction of data smaller than \( x \) is \( p \). A quantile function is actually the inverse of the cumulative distribution function of a random variable. I use the methods for finding quantiles in case of log-concave distribution and divide the domain into \( \gamma \) sub-intervals, \( I_1, \ldots, I_d \), such that \( P_{y_{\text{max}}}(\text{MAX} \in I_i) = \frac{1}{\gamma} \), for \( 1 \leq i \leq \gamma \). Note that in practice, for example, we can use the function \textit{quantile} in MatLab for this purpose, and there is a large set of distribution (e.g., Gaussian) for which there exists polynomial-time method to compute/estimate quantiles.

In Lemma 12 I show that if the predicate \( \text{AllSafe} \) is evaluated to 1 in case of the \( \gamma \) intervals for a smaller privacy parameter \( \tilde{\lambda} \) (i.e., stricter privacy requirement) then it is evaluated to 1 in case of the \( \omega \)-significant intervals as well.

**Lemma 12.** Suppose \( \text{Safe}_{\tilde{\lambda}, I} = 1 \) for each interval of the \( \gamma \) intervals, and \( \tilde{\lambda} = \frac{\lambda(c-1)-2}{c+1} \), \( c \) is any integer greater than \( 1+2/\lambda \). Then, \( \text{Safe}_{\lambda, I} = 1 \) for every \( \omega \)-significant interval \( J \).

**Proof.** (Sketch taken from section ?? of the Appendix A in my report [Th10 , 2012])

Based on the intuition I use during my proof (see the three cases discussed below) and to achieve that \( \tilde{\lambda} \) is smaller than \( \lambda \), I set \( \lambda \) such that \( \frac{\lambda}{c+1}(1+\tilde{\lambda}) = (1+\lambda) \). Further, to make \( \tilde{\lambda} \) be positive, based on the setting of \( \tilde{\lambda} \) above I choose the parameter \( c \) to be larger than \( 1+2/\lambda \).

In addition, \( \gamma \) is set to be larger than \( \omega \), namely, to \( [cw] \), where the brackets represent ceiling. Finally, let \( J \) be a \( \omega \)-significant interval and denote \( P(\text{MAX} \in J) = P_{y_{\text{max}}}^x \), and let \( d = [\gamma P_{y_{\text{max}}}^x] \). Note that with these settings of \( \gamma \) and \( d \) I have \( d \geq c \) and \( \frac{\lambda}{c+1} \leq \frac{c+1}{c} \).

My goal is to prove that the sequence \( \Lambda_i^x(q_i, a_i) \) is \( \lambda \)-Safe for each \( \omega \)-significant interval, and to do this, I prove a stronger privacy notion. Specifically, I show that if the sequence \( \Lambda_i^x(q_i, a_i) \) is \( \text{Safe}_{\tilde{\lambda}, I} = 1 \) for each interval \( I \), then it is \( (\frac{\lambda}{c+1})^2(1+\tilde{\lambda}) - 1 \)-Safe for every interval \( J \). This is a stronger privacy requirement because \( \frac{\lambda}{c+1}^2(1+\tilde{\lambda}) - 1 \leq \frac{\lambda}{c+1}(1+\tilde{\lambda}) - 1 = \lambda \). To prove this I examine three possible cases, and I show that this holds in all these cases: (Case 1) \( J \) is contained in the union of \( d+1 \) consecutive intervals, say \( I_1, I_2, \ldots, I_{d+1} \), of which \( J \) contains the intervals \( I_2, I_3, \ldots, I_{d+1} \); (Case 2) \( J \) is contained in the union of \( d+2 \) consecutive intervals, say \( I_1, I_2, \ldots, I_{d+2} \), of which \( J \) contains the intervals \( I_2, I_3, \ldots, I_{d+1} \); (Case 3) \( J \) is contained in the union of \( d+1 \) consecutive intervals, say \( I_1, I_2, \ldots, I_{d+1} \), of which \( J \) contains the intervals \( I_1, I_1, \ldots, I_{d+1} \).

Now I turn to the construction of the simulatable auditor. The construction of the simulatable auditor is as follows: According to the Definitions 25 and 26, first, I provide the method (Algorithm 3) for checking if the predicate \( \text{AllSafe} \) is 1 or 0, and thereafter I construct the simulatable auditor (Algorithm 4) based on the concept shown in the Figure 4.1 and the definition of \( (\lambda, \delta, \omega, T) \)-privacy game. Next, I going into details:

I give the algorithm \( \text{AllSafe} \), which is an estimation of the predicate \( \text{AllSafe}_{\lambda, \omega} \). This is because the algorithm makes use of the sampling algorithm \( \text{Sample}(G_{\tilde{\gamma}}, \epsilon) \) for estimating the posteriori probability, and instead of examining all the \( \omega \)-significant intervals, I make an estimation by only taking into account \( \gamma \) intervals: \( \text{AllSafe} \) takes as inputs

- the sequence of \( t \) queries and answers \( q_1, \ldots, q_t, a_1, \ldots, a_t \);
- \( G \), the distribution of each data \( x_i \);
- \( \eta \), which is a probability of error for computing \( \epsilon \),
- \( c \), which is the trade-off parameter such that \( \gamma = [cw] \), and \( \tilde{\lambda} = \frac{\lambda(c-1)-2}{c+1} \), where \( [\ ] \) represents ceiling.
- \( \omega \), which is the parameter of \( \omega \)-significant intervals.
- \( n \), the size of the data set.
The parameter choice is made such that the Lemma 13, which I discuss below, holds. In other words, if I modify the privacy parameters in Lemma 13 I have to modify the parameters above as well. Moreover, the intuition behind the parameter choice resides in the proof technique. In my proofs I apply the well-known definitions and theorems related to the Chernoff-bound, Union bound, and some basic statements in statistics and probability theory. Roughly speaking, these parameters have been chosen such that the Chernoff-bound and Union-bound can be applicable. I emphasize that the choice of these specific parameters is only for better illustrating purposes. These specific values of the parameters are one possible choice but not the only one. However, because the parameters strongly depend on each other, when modifying one parameter I have to accordingly modify the others. I also provide a general form of the parameters choice.

One drawback of the Lemma 12 is that the reverse direction is not necessarily true. Thus, to make claims on the AllSafe = 0 case, I cannot use directly the privacy parameter λ because the parameters strongly depend on each other, when modifying one parameter I have to modify the parameters above as well. Moreover, the intuition behind the parameter choice resides in the proof technique. In my proofs I apply the well-known definitions and theorems related to the Chernoff-bound, Union bound, and some basic statements in statistics and probability theory. Roughly speaking, these parameters have been chosen such that the Chernoff-bound and Union-bound can be applicable.

Algorithm 3: AllSafe (q1, . . . , qν, a1, . . . , aν, G, ω, λ, n, c)

Let AllSafe = TRUE;
for each of the γ intervals I in [α, β] do
    Sample N data sets according to G[i] using Sample(G[i], c);
    Let Nmax, Nmax ⊆ N, be the number of data sets for which MAX ∈ I;
    if \( \frac{2N_{max}}{N} \notin \left[ \frac{1}{1+\lambda'}, 1+\lambda' \right] \) then Let AllSafe = FALSE; endif
endfor
return AllSafe;

Algorithm 4: Simulatable probabilistic auditor

Inputs: q1, . . . , qν, a1, . . . , aν, a new query q, G, δ, η, λ, γ, n, T, c;
Let \( \epsilon = \frac{\delta}{10T} \);
for \( \frac{n}{\epsilon} \ln \frac{\gamma}{\gamma} \) times do
    Sample a consistent data set X′ according to G[i] using Sample(G[i], c);
    Let a′ = avgX′(Q); call AllSafe(q1, . . . , qν, a1, . . . , aν, G, ω, λ, n, c);
endfor
if the fraction of data sets X′ for which AllSafe=FALSE is greater than \( \frac{3\delta}{20\gamma} \) then
    return DENY; else return a′;
endif;

Intuitively, the steps in Algorithm 3 are as follows: By Lemma 12 instead of checking infinite ω-significant intervals with the privacy parameter λ we check the Safe predicate for each of the γ intervals and the smaller privacy parameter λ′. To estimate the posteriori probability that MAX ∈ I, we sample sufficient number (N) of data sets according to the distribution G[i], and compute the fraction (Nmax) of the data sets for which the maximum value falls in the interval I. Intuitively, by sampling according to G[i] we get the data sets that satisfy the condition λ′(avg(Q), i) = a′. If the ratio of the posteriori and apriori probabilities is outside the required bounds then the algorithm returns FALSE, otherwise TRUE is output.

Next I discuss how good estimation Algorithm 3 provides. In the ideal case, we would like that if the predicate AllSafe returns 0 (1) then the algorithm AllSafe returns FALSE (TRUE). However, we cannot make these claims for the next reasons: (i) we do not check all (infinitely many) ω-significant intervals for privacy and instead check only γ intervals; (ii) we estimate the
4.7. Simulatable auditor in the probabilistic disclosure model

posteriori probability using sampling, which has some error. Hence, instead of achieving the ideal case I provide the following claims:

**Lemma 13.** 1. If $\text{AllSafe}_{\lambda, \omega}(q_1, \ldots, q_t, a_1, \ldots, a_t) = 0$ then Algorithm $\overline{\text{AllSafe}}$ returns FALSE with probability at least $1 - \eta$.

2. If $\text{AllSafe}_{\lambda/m, \gamma}(q_1, \ldots, q_t, a_1, \ldots, a_t) = 1$ then Algorithm $\overline{\text{AllSafe}}$ returns TRUE with probability at least $1 - 2\gamma\eta$, where $m$ is such that $m > h$.

Intuitively, with probability close to 1, whenever $\text{AllSafe}_{\lambda, \omega} = 0$ the algorithm $\overline{\text{AllSafe}}$ also returns FALSE, and for a smaller privacy parameter $\lambda/m$ whenever $\text{AllSafe}_{\lambda/m, \gamma} = 1$ then $\overline{\text{AllSafe}}$ returns TRUE. The proof of the Lemma 13 can be found in the Appendix C in my report [Th10 , 2012].

The question is that beside these chosen parameters, how large should $N$ be, that is, how many data sets should be sampled? I showed that setting $N$ to be at least the maximum of

$$\left\{ \frac{4\gamma^2 \ln(2/\eta)(h + \lambda)^2(1 + \lambda)^2}{(h-1)^2\lambda^2}, \frac{4\gamma^2 \ln(2/\eta)(h + \lambda)^2(2 + \lambda)^2}{(m-h)^2\lambda^2} \right\}$$

is suitable for fulfilling the claims in the Lemma 13.

In the rest of the dissertation, however, I set $h = 3$ and $m = 9$ for easier discussion and illustrating purposes. This results in the following specific form of Lemma 13 as follows:

**Lemma 14.** 1. If $\text{AllSafe}_{\lambda, \omega}(q_1, \ldots, q_t, a_1, \ldots, a_t) = 0$ then Algorithm $\overline{\text{AllSafe}}$ returns FALSE with probability at least $1 - \eta$.

2. If $\text{AllSafe}_{\lambda/m, \gamma}(q_1, \ldots, q_t, a_1, \ldots, a_t) = 1$ then Algorithm $\overline{\text{AllSafe}}$ returns TRUE with probability at least $1 - 2\gamma\eta$.

**Proof.** (Sketch, taken from he Appendix B in my report [Th10 , 2012]) The proof and the parameter setting for this Lemma is based on the application of the well-known Chernoff-bound and Union-bound. Let $X_1, \ldots, X_n$ be independent Bernoulli trials (or Poisson trials), with $P(X_i = 1) = p$ (or $P(X_i = 1) = p_i$ in case of Poisson trials). Let $X = \sum_{i=1}^{n} X_i$ with $\mu = E[X]$, and $\theta \in (0, 1]$. The Chernoff-bound says: $P(X \leq \mu(1 - \theta)) \leq e^{-\theta\mu^2/2} \leq e^{-\theta\mu^2/4}$, and $P(X \geq \mu(1 + \theta)) \leq e^{-\theta\mu^2/4}$. The Union-bound says that if we have the events $e_1, \ldots, e_n$ then by applying the Chernoff-bound we can give a bound for the union of these events, that is, $P[e_1 \cup \cdots \cup e_n] \leq \sum_{i=1}^{n} P[e_i] \leq \sum_{i=1}^{n} \text{bound}_i$.

Intuitively, with probability close to 1, whenever $\text{AllSafe}_{\lambda, \omega} = 0$ the algorithm $\overline{\text{AllSafe}}$ also returns FALSE, and for a smaller privacy parameter $\lambda/9$ whenever $\text{AllSafe}_{\lambda/9, \gamma} = 1$ then $\overline{\text{AllSafe}}$ returns TRUE. For the region in between, no guarantees can be made. Note that in the general case, by choosing properly the input parameters, in the second point of the Lemma, we can choose any privacy parameter smaller than $\lambda$.

Now that we have an algorithm that evaluates the predicate $\text{AllSafe}_{\lambda, \omega}$, I turn to discuss the construction of the simulatable auditor itself. During the auditor construction, besides making use of the algorithm $\overline{\text{AllSafe}}$ I also take into account the notion of the T-round privacy game. The Algorithm 4 that implements the $(\lambda, \delta, \omega, T)$-private simulatable auditor. In Algorithm 4, beyond the parameters used in $\overline{\text{AllSafe}}$, additional parameters $\delta$ and $T$ are concerning the $(\lambda, \omega, T)$-privacy game and the $(\lambda, \delta, \omega, T)$-privacy auditor, and $\epsilon$ is the sampling error. Intuitively, the auditor repeatedly samples, according to the distribution $g_{t-1}^n$, a data set $X'$ that is consistent with the previous $t-1$ queries and answers. Then the corresponding answer $a'_t$ is computed based on $X'$ and the query set $Q_t$ of the query $q_t$. Thereafter, I call the algorithm AllSafe with the previous queries and answers, along with $q_t$ and $a'_t$. If the fraction of data sets for which $\overline{\text{AllSafe}}$ returns FALSE is larger than $9\delta/20T$ then the auditor denies, otherwise it returns the true answer $a_t$. The reason of choosing $9\delta/20T$ is that I want to fulfill the definition of $(\lambda, \delta, \omega, T)$-privacy auditor. The proof that Algorithm 4 implements a $(\lambda, \delta, \omega, T)$-privacy auditor is based on the well-known theorems of the Chernoff bound and Union bound over T rounds of the privacy game.
Theorem 3. Algorithm 4 implements a \((λ, δ, ω, T)\)-private simulatable auditor, and its running time is \(N\gamma\frac{80T}{9\delta}\ln T\), where \(T\) is the running time of the algorithm Sample\((D_c, ϵ)\), and \(D_c\) represents either \(G^n_{t-1}\) or \(G^n_t\). Finally, the running time of the simulatable auditor after \(t\) queries is \(t\gamma N\frac{80T}{9\delta}\ln T\).

Proof. (Sketch from of the Appendix D in my report [Th10 , 2012]) Again, the proof of the first point is based on the Chernoff-bound and Union-bound. The running time results from the fact that we check \(γ\) intervals and sample \(N\) data sets in each of the \(\frac{80T}{9\delta}\) rounds, using the algorithm Sample. Finally, this process is executed totally \(t\) times after \(t\) queries.

The proof of the Theorem 3 can be found in the Appendix D of my report [Th10 , 2012]. Since the running time of the algorithm Sample is polynomial [46], the running time of the Algorithm 4 is polynomial. Assume that my simulatable auditor does not include the quantile computation procedure, however, note that there is a large class of \(G\) for which the quantile computation is polynomial-time.

4.8 Summary

In this chapter, I defined a novel setting for query auditing, where instead of detecting or preventing the disclosure of individual sensitive values, I want to detect or prevent the disclosure of aggregate values in the database. As a specific instance of this setting, in the dissertation, I studied the problem of detecting or preventing the disclosure of the maximum value in the database, when the querier is allowed to issue average queries to the database. I proposed efficient off-line and on-line query auditors for this problem in the full disclosure model, and an efficient simulatable on-line query auditor in the partial disclosure model. My related publication in this topic is the paper [Th09 , 2012].
Conclusion and future works

In this dissertation, I focus on security problems in different application fields of wireless sensor networks. I proposed formal and automated verification methods for analyzing the security of protocols designed for WSNs, as well as query auditing algorithms for protecting sensitive information in statistical databases.

In Chapter 2, I argued that designing secure ad-hoc network routing protocols requires a systematic approach which minimizes the number of mistakes made during the design. To this end, I proposed a variant of process algebra called the $sr$-calculus, which provides expressive syntax and semantics for analyzing at the same time (i.) cryptographic primitives and operations, (ii.) the nature of broadcast communication, and (iii.) the specification of node’s neighborhood in wireless medium, which are required for verifying secure routing protocols. I proposed a systematic and exhaustive proof technique for analyzing routing protocols with the $sr$-calculus.

In addition, I proposed a fully automatic verification method, called $sr$-verif, for secured ad-hoc network routing protocols, which is based on logic and a backward reachability approach. My method has a clear syntax and semantics for modeling secure routing protocols, and handles arbitrary network topologies. Finally, my method can be used to verify the security of source routing protocols when the network includes several attacker nodes, who can cooperate with each other, and run several parallel sessions of the protocol.

As for future work, I want to extend my verification method to analyze other classes of routing protocols such as distance vector and link state type routing protocols. In addition, I plan to improve the features of the developed verification tool.

In Chapter 3, I addressed the problem of formal and automated security verification of WSN transport protocols that may perform cryptographic operations. The verification of this class of protocols is difficult because they typically consist of complex behavioral characteristics, such as real-time, probabilistic, and cryptographic operations.

To solve this problem, I proposed a probabilistic timed calculus for cryptographic protocols, called $crypt_{prob}$ time, and demonstrated how to use it for proving security or vulnerability of protocols. To the best of my knowledge, this is the first such process calculus that supports an expressive syntax and semantics, real-time, probabilistic, and cryptographic issues at the same time. Hence, it can be used to verify systems that involve these three properties. For demonstration purposes, I applied $crypt_{prob}$ time to prove that both of the two previously proposed protocols, DTSN and SDTP, are vulnerable to the EAR flag setting attack, and the tricky sandwich attack. Taking into account the security holes in DTSN and SDTP, I proposed a new secured WSN transport protocol, called SDTP+, and proved that the discussed attacks against DTSN and SDTP do not work in SDTP+. Finally, I emphasize that although I only used $crypt_{prob}$ time to analyze WSN transport protocols, it can be applied to reasoning about other probabilistic, timed, cryptographic protocols as well.

In addition, I proposed an automated verification method, based on the PAT process analysis toolkit for this class of protocols, and used it to verify the security of the DTSN and SDTP.
protocols. To the best of my knowledge, PAT has not been used to verify WSN transport protocols before, however, I showed that it is well-suited for this purpose.

In the future, I focus on improving the automatic security verification for this class of systems/protocols. Currently I found that PAT is the most suitable tool because it enables us to define concurrent, non-deterministic, real time, and probabilistic behavior of systems in a convenient way. However, in its current form it does not support (or only in a very limited way) cryptographic primitives and operations, as well as the behavior of strong (external or insider) attackers. Finally, I believe that my proposed methods can be applied for verifying other similar systems, which I will show in my follow up work.

In Chapter 4, I defined a novel setting for query auditing, where instead of detecting or preventing the disclosure of individual sensitive values, I want to detect or prevent the disclosure of aggregate values in the database. As a specific instance of this setting, in the dissertation, I studied the problem of detecting or preventing the disclosure of the maximum value in the database, when the querier is allowed to issue average queries to the database. I proposed efficient off-line and on-line query auditors for this problem in the full disclosure model, and an efficient simulatable on-line query auditor in the partial disclosure model.

My future work is concerned with looking at other instances (e.g., other types of aggregates in the queries) and prototypical implementation of my algorithms for experimentation. Currently I have implemented in MATLAB the proposed offline and online query auditors for the full disclosure model. For the online online query auditor for the full disclosure model, I will examine in more details, how the tolerance threshold $t_\text{lr}$ affects the utility.
List of publications


Bibliography


