Formal verification of secure ad-hoc network routing protocols using deductive model-checking[*]

Ta Vinh Thong
thong@crysys.hu

Budapest University of Technology and Economics,
Department of Telecommunications,
Laboratory of Cryptography and System Security (CrySyS)

TECHNICAL REPORT

[*]This report is the extended and revised version of the 6 pages conference paper [8]

Contents

Abstract

Ad-hoc networks do not rely on a pre-installed infrastructure, but they are formed by end-user devices in a self-organized manner. A consequence of this principle is that end-user devices must also perform routing functions. However, end-user devices can easily be compromised, and they may not follow the routing protocol faithfully. Such compromised and misbehaving nodes can disrupt routing, and hence, disable the operation of the network. In order to cope with this problem, several secured routing protocols have been proposed for ad-hoc networks. However, many of them have design flaws that still make them vulnerable to attacks mounted by compromised nodes. In this paper, we propose a formal verification method for secure ad-hoc network routing protocols that helps increasing the confidence in a protocol by providing an analysis framework that is more systematic, and hence, less error-prone than the informal analysis. Our approach is based on a new process algebra that we specifically developed for secure ad-hoc network routing protocols and a deductive proof technique. The novelty of this approach is that contrary to prior attempts to formal verification of secure ad-hoc network routing protocols, our verification method can be made fully automated, and provides expressiveness for explicitly modelling cryptographic privitives.

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.

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 hand 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., [13] 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 against 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, in this paper, we propose a formal method to verify the correctness of secure ad-hoc network routing protocols. The examples presented in this paper mainly consider secure on-demand source routing protocols. Our approach is based on a new variant of process algebra, called the *sr*-calculus, which we specifically developed for modeling the operation of secure ad-hoc network routing protocols, and a proof technique based on deductive model checking. The systematic nature of our method and its well-founded semantics ensure that one can have much more confidence in a security proof obtained with our 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 [9, 2, 3, 4], the novelty of our approach is that it can be fully automated. We also propose a novel automated verification method, called *sr*-verif, for secure on-demand source routing protocols.

2 Some attacks against secure routing protocols

Several "secure" routing protocols have been proposed in the recent past for wireless ad hoc networks. However, later most of them are turned out to be vulnerable to various attacks. In this section, we introduce some of these attacks that serves as the motivation of our work. First, we discuss the attack called as *relay attack*, which has already been modelled in the previous works [16, 5, 11]. Afterwards we give an overview of more subtle attacks against the SRP and the Ariadne protocols [9, 3]. We emphasize that these kinds of attacks cannot be modelled directly and conveniently in the previous works [16, 11, 22]. Our emphasis is deliberately on modelling and verifying these kinds of attacks.

2.1 Relay attacks

Relay attacks are such kind of attacks in which the attacker node forwards the received message unchanged. A typical relay attack scenario against on-demand source routing protocols is shown in the Figure 1. In this scenario the attacker node A receives the request message that includes the route specified by the list of node IDs $[\ldots, N]$. At this point, node A should append its own identifier to the ID list above and forwards the request containing the ID list $[\ldots, N, A]$ to the node M. However, instead the attacker forwards the message unchanged to M. Then, node M appends it own identifier to the ID list and forwards it. This procedure continues until the request message reached the destination node. At this point after making verification steps the destination node sends back a reply message. The reply message propagates backward along the route in the request message. After a while the reply reaches node M and is forwarded "backward". Due to the wireless environment and the fact that the attacker node is within the transmission range of node M, the reply is intercepted by the attacker. When the attacker node receives the reply message it forwards it unchanged. At the end, the source node accepts the route $[\ldots, N, M, \ldots]$ which does not exist in the current topology.

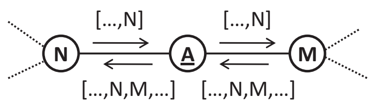


Figure 1: A typical relay attack scenario against on-demand source routing protocols.

2.2 An attack against the SRP protocol

SRP is a secure on-demand source routing protocol for ad-hoc networks proposed in [17]. The design of the protocol is inspired by the DSR protocol [15], 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; hence, SRP assumes only shared keys between communicating pairs.

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 basic problem in 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. 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.

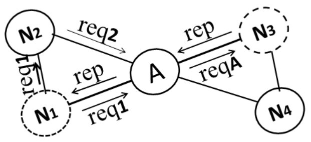


Figure 2: An attack scenario against the SRP protocol.

In order to illustrate a known attack on SRP, let us consider the network topology shown in Figure 2. Let us further assume that node N_1 initiates a route discovery to node N_3.

The attacker node A can manipulate the accumulated list of node identifiers in the route request such that N_3 receives the request with the list (N_2, λ, N_4), where λ is an arbitrary sequence of fake identifiers. This manipulation remains undetected, because the MAC computed by N_1 does not protect the accumulated node list in the route request, and intermediate nodes do not authenticate the request. When the target N_3 sends the route reply, A forwards it without modification to N_1 in the name of N_2. As the route reply is not modified, the MAC of the target N_3 verifies correctly at N_1, and thus, N_1 accepts the route (N_1, N_2, λ, N_4, N_3). However, this is a mistake, because there is no route via nodes N_2, λ, N_4.

2.3 An attack against the Ariadne protocol

In this subsection, we show the attack has been found in [9] against the Ariadne secure routing protocol.

Ariadne has been proposed in [14] 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 [18], on digital signatures, or on MACs. We 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 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. When the target receives the request, it verifies the perhop hash by re-computing the initiators MAC and the perhop 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.

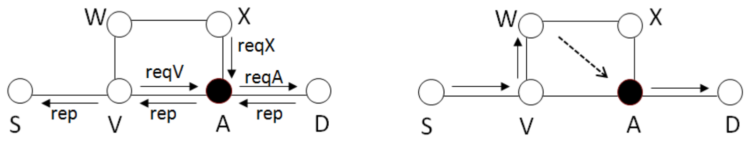


Figure 3: A subtle attack against Ariadne. The figure on the left shows the communication during the route discovery, while the figure on the right illustrates that at the end of the route discovery phase, the source node accepts the route S, V, W, A, D, which is not valid because the link between W and A does not exist.

Let us consider Figure 3, 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:

$$reqV = (rreq,\ S,\ D,\ ID,\ h_V,\ (V),\ (sig_V))$$

where ID is the random request identifier, h_V is the per-hop hash value generated by V, and sig_V is the signature of V.

After receiving req_V the attacker waits for an another route request from X:

$$reqX = (rreq,\ S,\ D,\ ID,\ h_X,\ (V,W,X),(sig_V,\ sig_W,\ sig_X)).$$

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_1, and H is the publicly known hash function used in the protocol. A obtains the signatures sig_V, sig_W from req_2. Then, A generates and broadcasts the following request:

$$reqA = (rreq,\ S,\ D,\ ID,\ h_A,\ (V,W,A)\ (sig_V,\ sig_W,\ sig_A))$$

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.

2.4 Summary

To sum up this section, we note that the modelling of the relay attacks has been addressed in some related works [11, 16]. However, relay attacks are mainly concerned in *neighbor discovery* [19] instead of route discovery. In contrast, in our work we primarily focus on formal verification of the attacks concerning *routing*, such as the attack against the SRP and Ariadne protocols we shown above. We note that modelling the attacks against the SRP and Ariadne protocols is more difficult than modelling relay attacks because they required the modelling of more complex attackers.

3 Motivation and Related works

As we can see, the discussed attacks are very subtle, thus, it is hard to detect and reasoning about them manually. In addition, reasoning in the hand and pencil manner is error-prone, therefore a systematic and automatic method is required.

Our purpose in this paper is to provide a formal modelling of secure on-demand source routing protocols and a systematic and automatic method for detecting attacks similar to the attacks we introduced in the previous section. Each method proposed in the most important related works [11, 12, 22, 5, 16, 9, 3, 23, 21] has numerous drawbacks that we will discuss in the following:

In works [9, 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 arbitrary feasible attacks. The 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. In this paper, we aim at improving these works by adding automated verification method based on deductive model-checking.

In order to give a formal and precise mathematical reasoning of the operation of routing protocol for wireless ad-hoc networks several process calculi have been proposed in the recent years. Among them the two works [11, 22] are closest to our work.

In [11] the author proposes the process algebra, called CMAN, for formally modelling the operarion of mobile ad-hoc networks. The advantage of this method is that it provides the modelling of cryptographic primitives and it is focused mainly on modelling mobility nature of mobile ad-hoc networks. The drawback of this method is that it cannot be directly used for modelling such attacks as the attacks scenario against the Ariadne or SRP protocols showed in the Section 2. In the attack scenario against the Ariadne protocol the attacker node waits, collects information it intercepts from the neighbor nodes and use them to construct an message that contains an incorrect route. CMAN does not provide syntax and semantics for modelling a knowledge base of the attacker node. In order to directly model these kind of attacks we propose the notion of the *active substitution with range* in the *sr*-calculus.

In the work [22] the authors propose the process algebra that is called as the ω-calculus. The main advantage of this calculus is that it provides the explicit modelling elements of broadcast communication and mobility. The main drawback of this method that it does not provide the syntax and semantics for modelling cryptographic primitives and attacker's knowledge base. In contrast to these our calculus provides the modelling of cryptographic primitives and attacker accumulated knowledge.

The advantage of these process algebras 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.

Several works in the literature address the problem of automatic verification of routing protocols. In the works [23, 21] the authors investigate the problem of verifying loop freedom property of routing protocols. In [23] the LUNAR protocol is verified using the SPIN, and UPPAAL model-checkers; in [23] the authors verified the DYMO protocol using graph transformation. In contrast to these works we propose an automatic verification method focuses on verifying security properties of "secure" routing protocols instead of loop freedom property.

The two works that are most related to our work are [5, 16]. In the work [5], and [16] the authors address the problem of verification security properties of secure routing protocols using the SPIN model-checker and CPAL-ES tools, respectively.

The main drawbacks of these methods are that they are based on the formal languages with a limited syntax and semantics. In particular, they cannot directly model cryptographic primitives and boadcast communications, instead they simulate cryptographic primitives with a series of bytes [5] and broadcast communication with a sequence of unicast communication. In contrast to these works, our automatic verification method provides the modelling elements for cryptographic primitives and neighborhood.

4 The *sr*-calculus

In this section we define the proposed *sr*-calculus: its syntax and informal semantics, as well as its operational semantics. We call this calculus as *sr*-calculus, where the prefix *sr* refers to the words **s**ecure **r**outing.

The advantage of the *sr*-calculus is that it provides expressiveness for modelling broadcast communication, neighborhood, mobility, and transmission range like in CMAN [11] and the ω-calculus [22], and the explicit modelling of cryptographic primitives like in the applied π-calculus [10], however, compared to them it includes the novel definition of *active substitution with range* that enables us to model attacker knowledge and attacks in the context of wireless ad-hoc networks.

CMAN cannot be used to directly model the attacks we found against the well-known SRP and Ariadne protocols [9, 3]. In these attacks the attacker can receive information from several paths, the attacker node then collects and store these information and construct a message that contains an incorrect route.

The ω-calculus lacks of modelling cryptographic primitives, such as digital signature and hashing, and has been used for modelling loop-freedom properties of AODV. Hence, it also lacks sytax and semantics for modelling attacker's knowledge set. In addition, neighbor nodes is organized into groups. However, it is not easy to determine groups in the topology.

Finally, the applied pi-calculus [10] provides active substitution that can be used to model actual attacker's knowledge. However, it lacks syntax and semantics for broadcasting, neighbors and deals with Dolev-Yao attacker model, which is not true in case of MANET.

We combine the advantage of each in order to provide a calculus with which we can directly and conveniently modelling and proving security properties and reasoning about attacks discussed in Section 2.

4.1 Type system of the *sr*-calculus

In this subsection, we provide a basic type system for the *sr*-calculus. The main purpose of the type system is to reduce the number of the possible cases to be examined during formal security proofs. Based on the type system we are capable of capturing errors such as arity mismatch and errounous binding/substitution of terms. We adapt the type system proposed for the applied π-calculus, discussed in the chapter 4 of [7], 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. In this paper, we only provide a brief overview of the type system in the chapter 4 of [7].

The type system catches the errors such as arity errors and matching of terms of different type. The type system does not include recursive types, so the type of processes such as $\overline{c}\langle c\rangle.P$ is not defined.

4.2 Basic definitions and terminology

Definition 1. *Type assignment is an assignment $v : T$ (or $u : T$) of a type T to v (or u) that can be a name, a constant, a node ID or a variable.*

Definition 2. *A well-formed type environment Γ is a finite set of type assignments where all names and variables are distinct. The domain of Γ is $dom(\Gamma) = \{\ v \mid \exists\ T.\{\ v : T\} \in \Gamma\ \}$*

Definition 3. *Let Γ and Δ type environments. We say that Γ extends Δ if the following holds:*

- $dom(\Gamma) \subseteq dom(\Delta)$

- *if $v \in dom(\Delta)$ such that $\{v : T\} \in \Delta$, then $\{v : T\} \in \Gamma$.*

If $\Delta = \{v_1 : T_1,\ldots,\ v_n : T_n\ \}$ and $\Gamma = \{u_1 : T_1,\ldots,\ u_n : T_n\ \}$ and $dom(\Gamma) \cap dom(\Delta) = \emptyset$, then Δ can be extended with Γ by taking their union, denoted by $\Delta \uplus \Gamma$.

4.3 Simple type system for the *sr*-calculus

The set of types is divided into the sets of *term types* and *process/behavior types*. Within the *term types* we distinguish among *channel types*, *broadcast types*, *name types*, *variable types*, *constant types*, and *node ID types*.

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

Let Γ be a type environment and Λ an expression which may be either a term, a process, or an extended process. A type judgment $\Gamma \vdash \Lambda : T$ is an assertion that the expression Λ has type T under the assumptions given in Γ. In particular, $\Gamma \vdash \Lambda : T$, asserts the following depending on Λ. If Λ is a term, then T is term type T_t. Thus $\Gamma \vdash t : T_t$ asserts that t has term type under the assumptions of Γ. The so called behavior type, denoted by T_{proc}, is introduced for processes. T_{pnetw} and T_{enetw} is the $\Gamma \vdash P : T_{proc}$ asserts that process P respects the type assertions in Γ. The judgment ($\vdash \Gamma$ *well-formed*) means that Γ is a well-formed type environment.

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

$$\begin{array}{rll}
S, T & ::= T_t \mid T_{proc} & (\textbf{Types}) \\
T_t & ::= T_{ch} \mid T_{br} \mid T_{str} & \text{(Term Types)} \\
T_{str} & ::= T_{name} \mid T_{var} \mid T_f \mid T_{const} & \text{(String Types)} \\
T_{name} & ::= tn_1 \mid \ldots \mid tn_n & \text{(Name Types)} \\
T_{var} & ::= tv_1 \mid \ldots \mid tv_n & \text{(Variable Types)} \\
T_f & ::= f(T^1_{str}, \ldots, T^n_{str}) & \text{(Function Types)} \\
T_{const} & ::= T_{req/rep} \mid tconst_i & \text{(Constant Types)} \\
T_{ch} & ::= \text{ch}(T^1_{str}, \ldots, T^n_{str}) & \text{(Channel Types)} \\
T_{br} & ::= \text{bch}(T^1_{str}, \ldots, T^n_{str}) & \text{(Broadcast Channel Types)} \\
T_{id} & ::= tl_1 \mid \ldots \mid tl_n & \text{(Node ID Types)} \\
T_{proc} & ::= tp_1 \mid \ldots \mid tp_n & \text{(Process/Behavior Types)} \\
T_{pnetw} & ::= tpnet_1 \mid \ldots \mid tpnet_n & \text{(Plain Network Types)} \\
T_{enetw} & ::= tenet_1 \mid \ldots \mid tenet_n & \text{(Extended Network Types)}
\end{array}$$

where *tn*, *tv*, *tl* and *tp*, *tpnet* and *tenet* are name types, variable types, node ID types, process types, plain and extended network types, respectively. The abbreviation of $x_1 : T_1,\ldots,\ x_n : T_n$ is defined by $\vec{x} : \vec{T}$. 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 direction 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 we 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 we distinguish types of each crypto function, such as, digital signature type, T_{sig}, one-way hash type, T_{hash}, MAC function type, T_{mac}. We also define types of secret key, T_{skey}, public key T_{pkey}, and symmetric shared key T_{shkey}. In this paper we only use these three crypto functions, 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.

$$\begin{array}{rll}
T_{skey} & ::= sk(T_{id}) & \text{(Secret Key Types)} \\
T_{pkey} & ::= pk(T_{id}) & \text{(Public Key Types)} \\
T_{shkey} & ::= k(T_{id}, T_{id}) & \text{(Shared Key Types)}
\end{array}$$

$$\begin{array}{rlr} T_{sig} ::= & sign(T_{str}, T_{skey}) & \text{(Digital Signature Types)} \\ T_{hash} ::= & hash(T_{str}) & \text{(One-Way Hash Types)} \\ T_{mac} ::= & mac(T_{str}, T_{key}) & \text{(MAC Types)} \end{array}$$

The syntax, the reduction rules and the trasition rules for the typed applied π-calculus remains unchanged from the ones defined for the untyped applied π-calculus. In order to ensure that structural equivalence preserves well-typedness, we require that the type system assigns equal types to terms that are equated by the equational theory.

Definition 4. *(Well formed environment)*

- $\vdash \emptyset$ *well-formed*
- *If* $(\vdash \Gamma$ *well-formed*$)$ *and* $u \notin dom(\Gamma)$, *then* $(\vdash \Gamma \uplus u$ *well-formed*$)$.

Definition 5. *(Type rules for terms) Let* $t \in \mathcal{T}$ *be a term,* T *a type, and* Γ *a well-formed type environment. Then the type judgment* $\Gamma \vdash_{\mathcal{T}} t : T$ *holds if it can be derived by application of one of the following rules.*

- *If* $\vdash \Gamma$ *well-formed and* $u : T \in \Gamma$ *then* $\Gamma \vdash_{\mathcal{T}} u : T_{str}$
- *If* $\Gamma \vdash_{\mathcal{T}} t_1 : T^1_{str} \ldots \Gamma \vdash_{\mathcal{T}} t_n : T^n_{str}$, *and the arity of* f *is* n, *then* $\Gamma \vdash_{\mathcal{T}} f(T^1_{str}, \ldots, T^n_{str}) : T_{str}$, *for each function name* f.

Definition 6. *(Type rules for plain processes) Let* $P \in \mathcal{P}$ *be a plain process,* Γ *a well-formed type environment, and* T_{proc} *the behavior type. Then the judgment* $\Gamma \vdash_{\mathcal{P}} P : T_{proc}$ *holds if it can be derived by application of one of the following rules.*

- *If* $(\vdash \Gamma$ *well-formed*$)$ *then* $(\Gamma \vdash_{\mathcal{P}} nil : T_{proc})$
- *If* $(\Gamma \vdash_{\mathcal{P}} P : T_{proc})$ *and* $(\Gamma \vdash_{\mathcal{P}} Q : T_{proc})$, *then* $\Gamma \vdash_{\mathcal{T}} P \mid Q : T_{proc}$.
- *If* $(\Gamma \vdash_{\mathcal{P}} P : T_{proc})$ *then If* $(\Gamma \vdash_{\mathcal{P}} !P : T_{proc})$
- *If* $(\Gamma \uplus \{u : T_{name}\} \vdash_{\mathcal{P}} P : T_{proc})$ *then* $(\Gamma \vdash_{\mathcal{P}} \nu u.P : T_{proc})$
- *If* $(\Gamma \vdash_{\mathcal{T}} t_1 : T_{str})$, $(\Gamma \vdash_{\mathcal{T}} t_2 : T_{str})$, $(\Gamma \vdash_{\mathcal{P}} P : T_{proc})$, $(\Gamma \vdash_{\mathcal{P}} Q : T_{proc})$ *then* $(\Gamma \vdash_{\mathcal{P}} [t_1 = t_2]\ P$ *else* $Q : T_{proc})$, *and also* $(\Gamma \vdash_{\mathcal{P}} [t_1 = t_2]\ P : T_{proc})$.
- *If* $(\Gamma \vdash_{\mathcal{T}} l : T_{id})$, $(\sigma$ *is the set of* l_i, *where* $\Gamma \vdash_{\mathcal{T}} l_i : T_{id}$, *for* $i \in \{1, \ldots, n\})$, $(\Gamma \vdash_{\mathcal{P}} P : T_{proc})$, $(\Gamma \vdash_{\mathcal{P}} Q : T_{proc})$ *then* $(\Gamma \vdash_{\mathcal{P}} [l \in \sigma]\ P$ *else* $Q : T_{proc})$, *and also* $(\Gamma \vdash_{\mathcal{P}} [l \in \sigma]\ P : T_{proc})$.
- *If* $(\Gamma \uplus \{\vec{x} : \vec{T}_{str}\} \vdash_{\mathcal{P}} P : T_{proc})$ *and* $(\Gamma \vdash_{\mathcal{T}} c : ch(\vec{T}_{str}))$ *then* $(\Gamma \vdash_{\mathcal{P}} c(\vec{x}).P : T_{proc})$.
- *If* $(\Gamma \vdash_{\mathcal{T}} \vec{t} : \vec{T}_{str})$ *and* $(\Gamma \vdash_{\mathcal{T}} c : ch(\vec{T}_{str}))$ *and* $(\Gamma \vdash_{\mathcal{P}} P : T_{proc})$ *then* $(\Gamma \vdash_{\mathcal{P}} \overline{c}\langle \vec{t}\,\rangle.P : T_{proc})$.
- *If* $(\Gamma \uplus \{\vec{x} : \vec{T}_{str}\} \vdash_{\mathcal{P}} P : T_{proc})$ *and* $(\Gamma \vdash_{\mathcal{T}} br : bch(\vec{T}_{str}))$ *then* $(\Gamma \vdash_{\mathcal{P}} br(\vec{x}).P : bch(\vec{T}_{str}))$.
- *If* $(\Gamma \vdash_{\mathcal{T}} \vec{t} : \vec{T}_{str})$ *and* $(\Gamma \vdash_{\mathcal{T}} br : bch(\vec{T}_{str}))$ *and* $(\Gamma \vdash_{\mathcal{P}} P : T_{proc})$ *then* $(\Gamma \vdash_{\mathcal{P}} \overline{br}\langle \vec{t}\,\rangle.P : T_{proc})$. *In the rest of the paper to represent broadcast sending/receiving we simply use* $\langle \vec{t} \rangle.P$, *and* $(x).P$ *by removing br from the beginning.*
- *If* $(\Gamma \vdash_{\mathcal{T}} t : T_{str})$ *and* $(\Gamma \vdash_{\mathcal{T}} x : T_{var})$ *then* $(\Gamma \vdash_{\mathcal{P}} \{t/x\} : T_{proc})$.

Definition 7. *(Type rules for plain networks) Let* $N \in \mathcal{N}_{net}$ *be a plain network,* Γ *a well-formed type environment, and* T_{pnetw} *the plain network type. Then the judgment* $\Gamma \vdash_{\mathcal{N}_{net}} N : T_{pnetw}$ *holds if it can be derived by application of one of the following rules.*

- *If* ($\vdash \Gamma$ *well-formed*) *then* ($\Gamma \vdash_{\mathcal{N}_{net}} 0_N : T_{pnetw}$)

- *If* ($\Gamma \vdash_{\mathcal{T}} l : T_{id}$), ($\Gamma \vdash_{\mathcal{P}} P : T_{proc}$) *then* ($\Gamma \vdash_{\mathcal{N}_{net}} \lfloor P \rfloor_l : T_{pnetw}$).

- *If* ($\Gamma \vdash_{\mathcal{T}} l : T_{id}$), ($\sigma$ *is the set of* l_i, *where* $\Gamma \vdash_{\mathcal{T}} l_i : T_{id}$, *for* $i \in \{1, \ldots, n\}$), ($\Gamma \vdash_{\mathcal{P}} P : T_{proc}$) *then* ($\Gamma \vdash_{\mathcal{N}_{net}} \lfloor P \rfloor_l^{\sigma} : T_{pnetw}$).

- *If* ($\Gamma \vdash_{\mathcal{N}_{net}} \lfloor P \rfloor_l^{\sigma_1} : T_{pnetw}$) *and* ($\Gamma \vdash_{\mathcal{N}_{net}} \lfloor P \rfloor_l^{\sigma_2} : T_{pnetw}$) *then* ($\Gamma \vdash_{\mathcal{N}_{net}} \lfloor P \rfloor_{l_1}^{\sigma_1} \mid \lfloor P \rfloor_{l_2}^{\sigma_2} : T_{pnetw}$).

- *If* ($\Gamma \uplus \{n : T_{name}\} \vdash_{\mathcal{N}_{net}} N : T_{pnetw}$) *then* ($\Gamma \vdash_{\mathcal{N}_{net}} \nu n.N : T_{pnetw}$).

  If ($\Gamma \vdash_{\mathcal{N}_{net}} N_1 : T_{pnetw}$) *and* ($\Gamma \vdash_{\mathcal{N}_{net}} N_2 : T_{pnetw}$) *then* ($\Gamma \vdash_{\mathcal{N}_{net}} N_1 \mid N_2 : T_{pnetw}$).

Definition 8. *(Type rules for extended networks) Let* $E \in \mathcal{E}_{net}$ *be an extended network,* Γ *a well-formed type environment, and* T_{pnetw} *the plain network type. Then the judgment* $\Gamma \vdash_{\mathcal{E}_{net}} E : T_{enetw}$ *holds if it can be derived by application of one of the following rules.*

- *If* ($\Gamma \vdash_{\mathcal{N}_{net}} N : T_{pnetw}$) *then* ($\Gamma \vdash_{\mathcal{E}_{net}} N : T_{enetw}$).

- *If* ($\Gamma \vdash_{\mathcal{E}_{net}} E_i : T_{enetw}$) *and* ($\Gamma \vdash_{\mathcal{E}_{net}} E_j : T_{enetw}$) *then* ($\Gamma \vdash_{\mathcal{E}_{net}} E_i \mid E_j : T_{enetw}$).

- *If* ($\Gamma \uplus \{n : T_{name}\} \vdash_{\mathcal{E}_{net}} E : T_{enetw}$) *then* ($\Gamma \vdash_{\mathcal{E}_{net}} \nu n.E : T_{enetw}$).

- *If* ($\Gamma \uplus \{x : T_{var}\} \vdash_{\mathcal{E}_{net}} E : T_{enetw}$) *then* ($\Gamma \vdash_{\mathcal{E}_{net}} \nu x.E : T_{enetw}$).

- *If* ($\Gamma \vdash_{\mathcal{T}} t : T_{str}$), ($\Gamma \vdash_{\mathcal{T}} x : T_{var}$) *and* ($\sigma$ *is the set of* l_i, *where* $\Gamma \vdash_{\mathcal{T}} l_i : T_{id}$, *for* $i \in \{1, \ldots, n\}$), *then* ($\Gamma \vdash_{\mathcal{E}_{net}} \lfloor \{t/x\}^{\sigma} \rfloor : T_{enetw}$).

4.4 Syntax and informal semantics of the *sr*-calculus

We assume an infinite set of *names* $\mathcal{N}$ and *variables* $\mathcal{V}$, where $\mathcal{N} \cap \mathcal{V} = \emptyset$. Further, we define a set of *node identifiers* (node ID) $\mathcal{L}$, where $\mathcal{N} \cap \mathcal{L} = \emptyset$. Each node identifier uniquely identifies a node, that is, no any node pair shares the same identifier.

We define a set of terms as

$$t ::= req \mid rep \mid true \mid ACCEPT \mid c, n, m, k \mid l_i, l_a \mid x, y, z, v, w \mid f(t_1, \ldots, t_k)$$

- *req* and *rep* are unique constants that represent the *req* and *rep* tags in route request and reply messages;

- *true* is a special constant that models the logical value 1;

- *ACCEPT* is a special constant. The source node outputs *ACCEPT* when it receives the reply message and all the verifications it makes on the reply are successful.

- c models communication channel;

- n, m and k are names and are used to model some data (e.g., a random nonce, a secret key);

- l_i, i can be numbers or letters not equal to a, and l_a are node IDs of the honest and the attacker node, respectively;

- x, y, z, v and w are variables that can represent any term, that is, any term can be bound to variables.

- Finally, f is a constructor function with arity k and is used to construct terms and to model cryptographic primitives, route request and reply messages. For instance, digital signature is modelled by the function $sign(t_1, t_2)$, where t_1 models the message to be signed and t_2 models the secret key. Route request and reply messages are modelled by the function *tuple* of k terms: *tuple*(t_1,..., t_k), which we abbreviate as (t_1,..., t_k).

We note that this definition of term is new compared to CMAN, the ω-calculus, and the applied π-calculus in that it includes the constants rep, req for modelling route discovery protocols, and process $[l \in \sigma]P$ for examining the neighborhood.

The internal operation of nodes is modelled by *processes.* Processes can be specified with the following syntax, and inductive definition:

P, Q, R ::=	processes
$\overline{c}\langle t\rangle.P$	unicast send
$c(x).P$	unicast receive
$\langle t\rangle.P$	broadcast send
$(x).P$	broadcast receive
$P\vert Q$	composition
$\nu n.P$	restriction
$!P$	replication
$[t_i = t_j]P$	if
$[t_i = t_j]P$ *else* Q	if-else
$[l \in \sigma]P$	in
$[l \in \sigma]P$ *else* Q	in-else
$\mathbf{0}$	nil
let $(x = t)$ *in* P	let

We note that this definition of processes is novel compared to the ω-calculus in that it includes constructor and destructor applications. Constructor/destructor application is used to model cryptographic primitives. Compared to CMAN it includes also the posibility of unicast. Finally, it differs from the applied π-calculus in that it includes broadcast send and receive actions.

- The process $\overline{c}\langle t\rangle.P$ represents the sending of message t on channel c followed by the execution of P, and $c(x).P$ represents the receiving of some message and binds it to x in P. For example, the communication between $\overline{c}\langle t\rangle.P$ and $c(x).P$ can be described as the reduction step of the parallel composition, namely, $\overline{c}\langle t\rangle.P \mid c(x).P \longrightarrow P \mid P\{t/x\}$. These two processes model the unicast send and receive actions.

- The two processes $\langle t\rangle.P$, and $(x).P$ represent the broadcast send and receive. Compared to the unicast case they does not contain the channel, which intends to model the fact that there is no any specified target.

- 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. Given a family of processes $P_1, P_2, \dots P_k$, we write $\prod_{i\in 1\dots k} P_i$, or $\prod_{i\in\{1\dots k\}} P_i$ for their parallel composition $P_1|P_2|\dots|P_k$.

- A restriction $\nu n.P$ is a process that makes a new, private name n, and then behaves as P.

- A replication $!P$ behaves as an infinite number of copies of P running in parallel.

- Processes $[t_i = t_j]P$ and $[l \in \sigma]P$ mean that if $t_1 = t_2$ and $l \in \sigma$, respectively, then process P is "activated", else they get stuck and stay idle.

- Processes $[t_i = t_j]P$ *else* Q and $[l \in \sigma]P$ *else* Q mean that if $t_1 = t_2$ and $l \in \sigma$, respectively, then process P is "activated", otherwise, Q is "activated".

- The *nil* process $\mathbf{0}$ does nothing.

- Finally, *let* $(x = t)$ *in* P means that the procedure of binding t to free occurrences of x in process P.

A "physical" node is defined as $\lfloor P \rfloor_l^\sigma$, which represents a node with the identifier l behaves as P and its transmission range covers nodes with the identifiers in the set σ. Two nodes are neighbors if they are in each other's range. We note that σ can be empty, denoted as $\lfloor P \rfloor_l$, 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: $\lfloor P \rfloor_l$; the parallel composition of nodes: $\lfloor P_1 \rfloor_{l_1}^{\sigma_1} \mid \lfloor P_2 \rfloor_{l_2}^{\sigma_2}$, where σ_1 and σ_2 may include l_2 and l_1 respectively; a network with name restriction, and the parallel composition of networks: $N_1 \mid N_2$.

$$N ::= \mathbf{0}_N \mid \lfloor P \rfloor_l \mid (\lfloor P_1 \rfloor_{l_1}^{\sigma_1} \mid \lfloor P_2 \rfloor_{l_2}^{\sigma_2}) \mid \nu n.N \mid (N_1 \mid N_2)$$

Note that $\lfloor P \rfloor_{l_1}^{l_2}$ and $\lfloor Q \rfloor_{l_2}^{l_1}$ means the node l_1 and node l_2 are bidirectionally connected. $\nu n.N$ represents the creation of new name n, such as secret keys, a nonce and only N knows it.

We stress that we use the form $\lfloor P \rfloor_l^\sigma$ that is already proposed in CMAN, and not the definition of groups (where neighbor nodes is organized to the same group) as in ω-calculus because we found that the topology of Mobile Ad-hoc Networks is usually represented as a graph and is given in the form of an adjacency matrix, thus, using the groups method in ω-calculus an algorithm can be required which extracts the cliques in the graph. Moreover, there can be redundant groups that should be handled properly for efficiency purpose. Finally, we prefer the $\lfloor P \rfloor_l^\sigma$ form because it gives us a possibility to model uni-directional links, which is not the case in ω-calculus. However, we note that CMAN does not include syntax and semantics for uni-directional links.

In order to model attacker's knowledge base and modelling the attacks, where attacker waits, collects and stores information feasible and more convenient, we extend the definition of networks with *the substitution with range* and the restriction on variables. Additionally, we adapt the the notion of active substitution for modelling the attacker's actual knowledge.

Again, we emphasize that the notion of active substitution and static equivalence have been used in the applied π-calculus, however, it models the knowledge of a Dolev-Yao attacker who eavesdrops every message that has been sent by communicating partners without considering the attacker model in wireless ad-hoc networks.

We slightly modified the notion active substitution to model such an adversary who can only intercept messages sent by its neighbors. More precisely, the intercepting of broadcast information is restricted to only nodes in the broadcast range. Furthermore, we adapt the concept of active substitution for modelling the attacker's knowledge set, which can continually change during the protocol. Finally, we emphasize that this is novel compared to all of the three calculi: CMAN, the ω-calculus, and the applied π-calculus.

The definition of the *extended network* is as follows:

$E ::=$	extended network
N	plain network
$E_i \mid E_j$	parallel composition
$\nu n.E$	name restriction
$\nu x.E$	variable restriction
$\lfloor \{t/x\}^\sigma \rfloor$	active substitution with range

- N is a plain network we already discussed above.
- $E_i \mid E_j$ is a parallel composition of two extended networks.
- $\nu n.E$ is a restriction of the name n to E.
- $\nu x.E$ is a restriction of the variable x to E.

- $\lfloor \{t/x\}^{\sigma} \rfloor$, which is abbreviated as $\{t/x\}^{\sigma}$ in the rest of the paper: $\{t/x\}^{\sigma}$ means that the substitution $\{t/x\}$ is applied to any node that is in parallel composition with $\{t/x\}$ and its identifier is in the set σ. Intuitively, we can say that σ is the range of the substitution $\{t/x\}$. Formally, we can explain the notion of active substitution with range by $\nu x.\left(\{t/x\}^{\sigma} \mid \prod_{l_i \in \sigma} \lfloor Q_i \rfloor_{l_i}^{\sigma_i}\right)$. This formula in turn can be defined as $\nu x.(\{t/x\} \mid A^{\sigma})$, where A^{σ} is the extended process (the notion of extended process A is presented in the applied π-calculus) that only includes the internal behavior of nodes in σ, and $\{t/x\}$ is the active substitution known in the applied π-calculus.

We write $fv(E)$, $bv(E)$, $fn(E)$, and $bn(E)$ for the sets of free and bound variables and free and bound names of E, respectively. These sets are defined as follow:

$$fv(\{t/x\}^{\sigma}) \overset{def}{=} fv(t) \cup \{x\},\ fn(\{t/x\}^{\sigma}) \overset{def}{=} fn(t) \cup \{node\ IDs \in \sigma\}$$

$$bv(\{t/x\}^{\sigma}) \overset{def}{=} \emptyset,\ bn(\{t/x\}^{\sigma}) \overset{def}{=} bn(t)$$

An extended network is closed when every variable is either bound or defined by an active substitution with range.

In the applied π-calculus, a frame is an extended process built up from 0 and active substitutions of the form $\{t/x\}$ by parallel composition and restriction. Analogously, we follow this concept and we let the frame of network, denoted by φ_N, be an extended network built up from 0_N and active substitutions with range, $\lfloor \{t/x\}^{\sigma} \rfloor$ (or simply $\{t/x\}^{\sigma}$). Every extended network E can be mapped to a frame $\varphi_N(E)$ by replacing every plain network embedded in E with empty network $\mathbf{0}_N$. We distinguish the notations frame of network (φ_N) and frame of processes (φ) also known in case of the applied π-calculus.

For example, the frame of process E, where

$$E = \{t_1/x_1\}^{\sigma_1} \mid \{t_2/x_2\}^{\sigma_2} \mid \ldots \mid \{t_k/x_k\}^{\sigma_k} \mid \textstyle\prod_i \lfloor Q_i \rfloor_{l_i}^{\sigma_i}$$

is $\varphi_N(E) = \{t_1/x_1\}^{\sigma_1} \mid \{t_2/x_2\}^{\sigma_2} \mid \ldots \mid \{t_k/x_k\}^{\sigma_k}$.

The frame $\varphi_N(E)$ can be viewed as an approximation of the behavior of E that accounts for the *static knowledge* exposed by E to its environment, but not for E's dynamic behavior.

In the next section, we give the semantics of the calculus in order to reason about secure on-demand source routing protocols.

4.5 Semantics

First we define the structural equivalence relation which is used to simplify a process of large size to a smaller one that is the is equivalent to the original one. This relation is very important in proofs. We say that two processes are structurally equivelence, if they are identical up to structure.

4.5.1 The Structural Equivalence($\equiv$)

In particular, structural equivalence relation is defined as the least equivalence relation satisfying bound name, bound variable conversion (also called as α-conversion) and the following rules:

(Rules for Processes:)

(Struct P-α)	$P \equiv_{x \leftarrow y} Q;\ P \equiv_{n_1 \leftarrow n_2} Q$
(Struct P-Par1)	$P \mid \mathbf{0} \equiv P$
(Struct P-Par2)	$P_1 \mid P_2 \equiv P_2 \mid P_1$
(Struct P-Par3)	$(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$
(Struct P-Switch)	$\nu n_1.\nu n_2.P \equiv \nu n_2.\nu n_1.P$
(Struct P-Repl)	$!P \equiv P \mid !P$
(Struct P-Drop)	$\nu n.0 \equiv 0$

(Struct P-Extr)	$\nu n.(P\|Q) \equiv P\|\nu n.Q$ if $n \notin fn(P)$
(Struct P-Let)	$let\ x = t\ in\ P \equiv P\{t/x\}$
(Struct P-If1)	$[t = t]P \equiv P$
(Struct P-If2)	$[t_i = t_j]P \equiv 0\ (if\ t_i \neq t_j)$
(Struct P-In1)	$[l \in \sigma]P \equiv P\ (if\ l \in \sigma)$
(Struct P-In2)	$[l \in \sigma]P \equiv 0\ (if\ l$ is not in $\sigma)$

The meaning of each rule is the following:

- Struct P-α: P and Q are stuctural equivalent if Q can be obtained from P by renaming one or more bound names/variables in P, or vice versa. For instance, processes $(x).P$ and $(y).P$ are structural equivalent by renaming y to x. This is denoted by $\equiv_{x \leftarrow y}$.
- Struct P-Par1: The parallel composition with the nil process does not change anything, the result is the same as the original parallel composition.
- Struct P-Par2: The parallel composition is commutative.
- Struct P-Par3: The parallel composition is associative.
- Struct P-Switch: The restriction is commutative.
- Struct P-Drop: Restriction does not affect the nil process, thus, we can drop it.
- Struct P-Extrusion: We can drop the restriction from process P when P does not contain the restricted name as free name, that is, the restricted name does not occur in P.
- Struct P-Let: Both sides represent the binding of the term t to variable x in P.
- Struct P-If1, P-If2: If the two terms are the same then the execution of P begins, while if they are distinct then the process gets stuck.
- Struct P-In1, P-In2: If the node identifier l is in the set σ then the execution of P begins, otherwise the process P gets stuck and stays idle.

The next additional rules are valid to structural equivalence:

$$\frac{P \equiv Q, Q \equiv R}{P \equiv R} \qquad \frac{P \equiv P'}{P|Q \equiv P'|Q} \qquad \frac{P \equiv P'}{\nu n.P \equiv \nu n.P'}$$

The first one means structural equivelence relation is transitive: if $P \equiv Q$ and $Q \equiv R$ then $P \equiv R$; the second and third rules show that structural equivalence closed to replication and restriction. Similarly the rules for network can be defined:

(Rules for Networks:)

(Struct N-Par1)	$N\|\mathbf{0}_N \equiv N$
(Struct N-Par2)	$N_1\|N_2 \equiv N_2\|N_1$
(Struct N-Par3)	$(N_1\|N_2)\|N_3 \equiv N_1\|(N_2\|N_3)$
(Struct N-Switch)	$\nu n_1 n_2.N \equiv \nu n_2 n_1.N$
(Struct N-Extr)	$(\nu n.N_1)\|N_2 \equiv \nu n.(N_1\|N_2)\ (if\ n \notin fn(N_2) \cup id(N_1))$
(Struct N-Node)	$\lfloor P \rfloor_l^\sigma \equiv \lfloor Q \rfloor_l^\sigma\ (if\ P \equiv Q)$
(Struct N-Rest)	$\lfloor \nu n.P \rfloor_l^\sigma \equiv \nu n.\ \lfloor P \rfloor_l^\sigma$

$fn(N)$ and $id(N)$ represent the set of free names of N and the set of node identifiers in N, respectively. The first five rules are standard, the only rules require some words to mention is (Struct N-Node) and (Struct N-Rest). The first rule means that two networks are structural equivalent if they contain the nodes with the same internal operation and having the same identifier with same neighbors. The second rule says that a name restriction on a plain process can be seen as a restriction on a node.

Again, we emphasize that this part is new compared to CMAN, and the ω-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 π-calculus in that active substitution has range for modelling neighborhood.

Finally, the rules for the extended network E are defined as follows:

(Rules for Extended Networks:)
(Struct E-Par1) $E|\mathbf{0}_N \equiv E$
(Struct E-Par2) $E_1|E_2 \equiv E_2|E_1$
(Struct E-Par3) $(E_1|E_2)|E_3 \equiv E_1|(E_2|E_3)$
(Struct E-Extr) $(\nu n.E_1)|E_2 \equiv \nu n.(E_1|E_2)$ $(if\ n \notin fn(E_2) \cup fv(E_2) \cup id(E_1))$
(Struct E-Switch) $\nu n_1 n_2.E \equiv \nu n_2 n_1.E$
(Struct E-Intro) $\nu x.\{t/x\}^\sigma \equiv \mathbf{0}_N$
(Struct E-Try) $\{t/x\}^\sigma|E \equiv \{t/x\}^\sigma|E\{t/x\}^\sigma$
(Struct E-Rewrite) $\{t_1/x\}^\sigma \equiv \{t_2/x\}^\sigma$ (if $t_1 = t_2$)

$fn(E)$, $fv(E)$ and $id(E)$ represents the set of free names of E, the set of free varibles of E, and the set of node identifiers in E, respectively. The first five rules is similar and come straightforward from the rules on networks and processes. Rule (Intro) is used to introduce any active substitutions. The rule (Struct E-Rewrite) say that two active subtitutions with the same range σ, and terms are stucturally equivalent. Rule (Try) represents the trying to apply substitution $\{t/x\}^\sigma$ to the extended network E: For example, let E be

$$E = \nu\tilde{n}(\{t_1/x_1\}^{\sigma_1} \mid \ldots \mid \{t_k/x_k\}^{\sigma_k} \mid \lfloor Q_i \rfloor_{l_i}^{\sigma_i} \mid \ldots \mid \lfloor Q_j \rfloor_{l_j}^{\sigma_j})$$

where $\tilde{n}$ is a collection of non-duplicated names. Then $E\{t/x\}^\sigma$, where $\tilde{n} \notin fn(t) \cup fv(t) \cup id(E)$, is

$$\nu x.\nu\tilde{n}.(\{t_1/x_1\}^{\sigma_1} \mid \ldots \mid \{t_k/x_k\}^{\sigma_k} \mid \lfloor Q_i \rfloor_{l_i}^{\sigma_i} \{t/x\}^\sigma \mid \ldots \mid \lfloor Q_j \rfloor_{l_j}^{\sigma_j} \{t/x\}^\sigma)$$

Intuitively, this means that the substitution is applied on every plain network. However, this substitution successes at $\lfloor Q_i \rfloor_{l_i}^{\sigma_i}$ only in case location $l_i \in \sigma$, otherwise, it has no effect. This is formally defined by the rules (E-Try) and (E-Subst):

(Struct E-Try-1) $\lfloor Q_i \rfloor_{l_i}^{\sigma_i} \{t/x\}^\sigma \equiv_{l_i \in \sigma} \lfloor Q_i\{t/x\} \rfloor_{l_i}^{\sigma_i}$
(Struct E-Try-2) $\lfloor Q_i \rfloor_{l_i}^{\sigma_i} \{t/x\}^\sigma \equiv_{l_i \notin \sigma} \lfloor Q_i \rfloor_{l_i}^{\sigma_i}$
(Struct E-Subst-1) $\{t/x\}^\sigma | \lfloor Q_i \rfloor_{l_i}^{\sigma_i} \equiv_{l_i \in \sigma} \{t/x\}^\sigma | \lfloor Q_i\{t/x\} \rfloor_{l_i}^{\sigma_i}$
(Struct E-Subst-2) $\{t/x\}^\sigma | \lfloor Q_i \rfloor_{l_i}^{\sigma_i} \equiv_{l_i \notin \sigma} \{t/x\}^\sigma | \lfloor Q_i \rfloor_{l_i}^{\sigma_i}$

In the next subsection, we provide the reduction relation that are used to model the internal operation/computation of nodes, and to model a reduction step in case of networks.

4.5.2 Reduction relation ($\rightarrow$)

(Internal reduction rules for processes:)
(Red P-Let) $let\ x = t\ in\ P \rightarrow P\{t/x\}$
(Red P-If1) $[t = t]P \rightarrow P$
(Red P-If2) $[t_i = t_j]P \rightarrow 0$ $(if\ t_i \neq t_j)$
(Red P-In1) $[l \in \sigma]P \rightarrow P$ $(if\ l \in \sigma)$
(Red P-In2) $[l \in \sigma]P \rightarrow 0$ $(if\ l$ is not in $\sigma)$

The operations (i) binding a variable to a term in a process; (ii) checking the equality of two terms; (iii) checking the presence of a node identifier in a set of node identifier; and (iv) destructor computations such as checking digital signatures, are all internal operations of nodes. Next we introduce the internal reduction steps in a network. Internal (i.e., silent) steps that can be performed by nodes can be connection and disconnection, specifying the mobility of nodes:

(Reduction relations for mobility:)
(Red Connect) $\lfloor P \rfloor_{l_1}^{\sigma_1} \mid \lfloor Q \rfloor_{l_2}^{\sigma_2} \rightarrow_{\{l_1 \bullet\ l_2\}} \lfloor P \rfloor_{l_1}^{\sigma_1 l_2} \mid \lfloor Q \rfloor_{l_2}^{\sigma_2}$, where l_2 is in σ_1.
(Red Disconnect) $\lfloor P \rfloor_{l_1}^{\sigma_1 l_2} \mid \lfloor Q \rfloor_{l_2}^{\sigma_2} \rightarrow_{\{l_1 \circ\ l_2\}} \lfloor P \rfloor_{l_1}^{\sigma_1} \mid \lfloor Q \rfloor_{l_2}^{\sigma_2}$, where l_2 is not in σ_1.

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

We note that although we have defined the rules for mobility, we will not use them in our proofs because we are only considering the analysis of the attacks discussed in Secition 2. Hence, we always assume that ***the topology does not change*** *during the attack.*

In order to reason about secure on-demand source routing protocols, and describing the operation semantics of the *sr*-calculus we introduce the labelled transition system of the calculus in the following subsection. Labelled transition system is very important in proofs, it captures such activities made by nodes that can be observed by the environment. This action, for instance, can be a broadcast sending or a broadcast receiving.

4.5.3 Labelled transition system ($\xrightarrow{\alpha}$)

We note that broadcast and unicast sending are non-deterministically executed, while receiving actions can take place at the same time.

The traditional operation semantics for processes is defined as a labelled transition system $(\mathcal{P}, \mathcal{G}, \rightarrow)$ where $\mathcal{P}$ represents a set of (plain) processes, $\mathcal{G}$ is a set of labels, and $\rightarrow \subseteq \mathcal{P} \times \mathcal{G} \times \mathcal{P}$. In our case it is the ternary relation $(\mathcal{E}_{net}, \mathcal{G}, \rightarrow)$ where $\mathcal{E}_{net}$ represents a set of extended networks, $\mathcal{G}$ is a set of labels, and $\rightarrow \subseteq \mathcal{E}_{net} \times \mathcal{G} \times \mathcal{E}_{net}$. The following labelled transitions are specified in this transition system in the *sr*-calculus.

(Labelled transition rules for networks)
(Ext BroadSend1) $\nu\tilde{n}.\, \lfloor \langle t \rangle.P \rfloor_{l}^{\sigma} \xrightarrow{\nu x.\langle x \rangle:\bar{l}\{\sigma\}} \nu\tilde{n}.\, (\{t/x\}^{\sigma} \mid \lfloor P \rfloor_{l}^{\sigma})$
(Ext BroadSend2) $\lfloor \langle t \rangle.P \rfloor_{l}^{\sigma} \xrightarrow{\nu x.\langle x \rangle:\bar{l}\sigma} (\{t/x\}^{\sigma} \mid \lfloor P \rfloor_{l}^{\sigma})$
(Ext BroadRecv1) $\lfloor (x).Q \rfloor_{l}^{\sigma_2} \xrightarrow{(t)^{\sigma}:\{l \in \sigma\}} \lfloor Q\{t/x\} \rfloor_{l}^{\sigma_2}$
(Ext BroadRecv2) $\nu\tilde{n}.\, \lfloor (x).Q \rfloor_{l}^{\sigma_2} \xrightarrow{(t)^{\sigma}:\{l \in \sigma\}} \nu\tilde{n}.\, \lfloor Q\{t/x\} \rfloor_{l}^{\sigma_2}$

New names $\tilde{n}$ typically represent some secret key or nonce in secure on-demand source routing protocols. The rules (Ext BroadSend1-2) say that the node l has broadcast term t, hence, it is now available for nodes in its range, σ. This is modelled by $\{t/x\}^{\sigma}$ and νx, which restricts the substitution to nodes within the range. The rules (Ext BroadRecv1-2) say that if the listening node l is within σ (which is the range of the node that sent t, denoted by $(t)^{\sigma}$) then it obtains t.

4.6 The connection between the *sr*-calculus and the applied π-calculus

In this subsection, we interpret the semantics of the processes and the communication between nodes in the *sr*-calculus based on the semantics of the processes in the applied π-calculus.

- Processes $\overline{c}\langle t \rangle.P$ and $c(x).P$ are interpreted as in case of the applied π-calculus.

- Process $\langle t \rangle.P$ can be interpreted as the process $\overline{c_{nbrofN}}\langle t \rangle.P$ in the applied π-calculus, where channel c_{nbrofN} is the communication channel shared between the broadcasting node N and its neighbors.
- Process $(x).P$ can be interpreted as the process $c_{nbrofN}(x).P$ in the applied π-calculus, where c_{nbrofN} is the communication channel shared between the broadcasting node and its neighbors. The non-neighbor nodes of N do not posses c_{nbrofN}. Hence, they will not receive the broadcast message.
- Process $[t_i = t_j]P$ can be interpreted by the process *if* $(t_i = t_j)$ *then* P in the applied π-calculus.
- Process $[t_i = t_j]P$ *else* Q can be interpreted by the process *if* $(t_i = t_j)$ *then* P *else* Q in the applied π-calculus.
- Process $[l \in \sigma]P$ can be interpreted by the process *if* $(l = l_1)$ *then* P *else if* $(l = l_2)$ *then* P *else* ... *if* $(l = l_m)$ *then* P, in the applied π-calculus, where $\sigma = \{l_1, \ldots, l_m\}$.
- Process $[l \in \sigma]P$ *else* Q can be interpreted by the process *if* $(l = l_1)$ *then* P *else if* $(l = l_2)$ *then* P *else* ... *if* $(l = l_m)$ *then* P *else* Q, in the applied π-calculus, where $\sigma = \{l_1, \ldots, l_m\}$.

The communication among nodes in the *sr*-calculus can be interpreted as follows in the applied π-calculus:

- $\lfloor \bar{c}\langle t \rangle.P \rfloor_{l_1}^{\sigma_1} \mid \lfloor c(x).Q \rfloor_{l_2}^{\sigma_2}$ can be interpreted as follows: $\bar{c}\langle t \rangle.P \mid R$, where R is as follows:

  $$\textit{if } (l_2 = l_{i_1}) \textit{ then } c(x).Q \textit{ else if } (l_2 = l_{i_2}) \textit{ then } c(x).Q \textit{ else} \ldots \textit{if } (l_2 = l_{i_m}) \textit{ then } c(x).Q \textit{ else } c(x).c(x).Q$$

  in the applied π-calculus, where $\sigma_1 = \{l_{i_1}, \ldots, l_{i_m}\}$. Intuitively, if $(l_2 \in \sigma_1)$ then $\bar{c}\langle t \rangle.P \mid R$ becomes $\bar{c}\langle t \rangle.P \mid c(x).Q$, while in case $(l_2 \notin \sigma_1)$ $\bar{c}\langle t \rangle.P \mid R$ becomes $\bar{c}\langle t \rangle.P \mid c(x).c(x).Q$. The rationale behind the process $\bar{c}\langle t \rangle.P \mid c(x).c(x).Q$ is that when $l_2 \notin \sigma_1$ node l_2 will not receive t sent by l_1 on channel c, hence, $\lfloor \bar{c}\langle t \rangle.P \rfloor_{l_1}^{\sigma_1} \mid \lfloor c(x).Q \rfloor_{l_2}^{\sigma_2}$ becomes $\lfloor P \rfloor_{l_1}^{\sigma_1} \mid \lfloor c(x).Q \rfloor_{l_2}^{\sigma_2}$. This is simulated by the applied π-calculus process $\bar{c}\langle t \rangle.P \mid c(x).c(x).Q$, because after sending t, the result process will be $P \mid c(x).Q\{t/x\}$, which is equivalent to $P \mid c(x).Q$. The reason is that according to the semantics of the applied π-calculus, $c(x).Q\{t/x\}$ only binds t to the free occurrence of x in $c(x).Q$, however, x is a bound variable in $c(x).Q$, hence, the binding $\{t/x\}$ will not apply.

- $\lfloor \langle t \rangle.P \rfloor_{l_1}^{\sigma_1} \mid \lfloor (x).Q \rfloor_{l_2}^{\sigma_2}$ can be interpreted as follows: $\overline{c_{nbrN1}}\langle t \rangle.P \mid R$, where c_{nbrN1} is the channel shared among node $N1$, $N1 = \lfloor \langle t \rangle.P \rfloor_{l_1}^{\sigma_1}$, and its neighbors, and R is defined as follows:

  $$\textit{if } (l_2 = l_{i_1}) \textit{ then } c_{nbrN1}(x).Q \textit{ else if } (l_2 = l_{i_2}) \textit{ then } c_{nbrN1}(x).Q \textit{ else} \ldots \textit{if } (l_2 = l_{i_m}) \textit{ then } c_{nbrN1}(x).Q \textit{ else } c_{nbrN1}(x).c_{nbrN1}(x).Q$$

  in the applied π-calculus, where $\sigma_1 = \{l_{i_1}, \ldots, l_{i_m}\}$.

As for the rules for modelling the mobility of nodes, whenever two nodes $N1$ and $N2$ become neighbors of each other, the scope of the two channels c_{nbrN1} and c_{nbrN2}, defined by the restrictions constructs νc_{nbrN1} and νc_{nbrN2}, will be extended. After extending the scope of c_{nbrN1} to $N2$ and c_{nbrN2} to $N1$, $N1$ and $N2$ can use the channels for communications. By adjusting the scope of the channels, we can model the connection and disconnection of the nodes.

4.7 Equational Theory

The destructor applications (e.g., proposed in the spi-calculus), which basically is the inverse of functions, and are used to model verification computations on messages. Formally, it is the process *let* $(x = g(t_1, \ldots, t_n))$ *in* P *else* Q, which tries to evaluate $g(t_1, \ldots, t_n)$ if this succeeds, x is bound to the result and P is executed, otherwise, Q is executed. For instance, a typical destructor can be verification of digital signature as $\mathit{checksign}(\mathit{sign}(x, \mathit{sk}(y)), \mathit{pk}(\mathit{sk}(y)))$, where the constructor $\mathit{pk}(\mathit{sk}(y))$ represents the public key generated from the given secret key *sk(y)*.

To make the proofs and the system specification be more simpler, instead of using destructor applications, we use the notion of equational theory proposed in the applied π-calculus. An equational theory Eq is defined over the set of function symbols $\sum$. It contains a set of equations of the form $t_1 = t_2$, where terms t_1, t_2 are defined in $\sum$. Like the destructor application it allows us to capture relationships between terms defined in $\sum$. Equality modulo the equational theory, written $=_{Eq}$, is defined as the smallest equivalence relation on terms, that contains Eq and is closed under application of function symbols, substitution of terms for variables and bijective renaming of names [10]. For instance, *dec(enc(x, y),y)* $=$ *x* and *checksign(sign(x, y),pk(y))* $=$ *true* for a special constant *true*. Note that we write $=$ instead of $=_{Eq}$ for simplicity because in our case it is clear from the context.

4.8 Examples

Next we show the application of active substitution with range and the defined labelled transition system on some example networks.

4.8.1 Example for broadcasting and message loss

The first example network illustrated in the Figure 4 includes three nodes. In this simple network, node P is assigned the identifier l_1, node Q has identifier l_2 and node R has the ID l_3. Node P and node Q are neighbors, but node P and R are not. Thus, when P broadcasts message t, only Q receives t. The following labelled transitions model the procedure in which P broadcasts t, and only Q receives it.

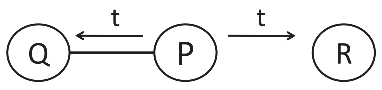


Figure 4: An example network

$$\Big(\lfloor\langle t\rangle.P_1\rfloor_{l_1}^{l_2} \mid \lfloor(y).Q_1\rfloor_{l_2}^{l_1} \mid \lfloor(z).R_1\rfloor_{l_3}\Big) \xrightarrow{\nu x.\langle x\rangle:\overline{l_1}\{l_2\}}$$
$$\Big(\{t/x\}^{\{l_1,l_2\}} \mid \lfloor P_1\rfloor_{l_1}^{l_2} \mid \lfloor(y).Q_1\rfloor_{l_2}^{l_1} \mid \lfloor(z).R_1\rfloor_{l_3}\Big) \xrightarrow{(t)^{\{l_1,l_2\}}:\{l_2\in\{l_1,l_2\}\}}$$
$$\Big(\{t/x\}^{\{l_1,l_2\}} \mid \lfloor P_1\rfloor_{l_1}^{l_2} \mid \lfloor Q_1\{t/x\}\rfloor_{l_2}^{l_1} \mid \lfloor(z).R_1\rfloor_{l_3}\Big).$$

This example includes only one broadcast step and there is no process replication. First, the rule (Ext BroadSend2) is applied, followed by (Ext BroadRecv1), which models the fact that Q receives t, because of $l_2 \in \{l_1, l_2\}$.

4.8.2 Example for multiple broadcast sends and receives

The next example is a little more complicated which also includes process replication and multiple broadcast sends. The network can be seen in the Figure 5. Here, both nodes P and Q are the neighbors of R. First P broadcasts t_1 then Q broadcasts t_2. Node R is under replication which intuitively means that it repeatedly listens for messages.

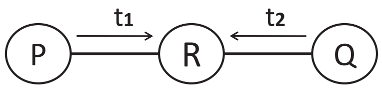


Figure 5: Another example network

$$
\begin{aligned}
&\Big(\lfloor\langle t_1\rangle.P\rfloor_{l_1}^{\{l_3\}} \mid \lfloor\langle t_2\rangle.Q\rfloor_{l_2}^{\{l_3\}} \mid \lfloor !(y).R\rfloor_{l_3}^{\{l_1,l_2\}}\Big) \xrightarrow{\nu x.\langle x\rangle:\overline{l_1}\{l_3\}} \\
&\Big(\{t_1/x\}^{\{l_1,l_3\}} \mid \lfloor P\rfloor_{l_1}^{\{l_3\}} \mid \lfloor\langle t_2\rangle.Q\rfloor_{l_2}^{\{l_3\}} \mid \lfloor !(y).R\rfloor_{l_3}^{\{l_1,l_2\}}\Big) \xrightarrow{(t_1)^{\{l_1,l_3\}}:\{l_3\in\{l_1,l_3\}\}} \\
&\Big(\{t_1/x\}^{\{l_1,l_3\}} \mid \lfloor P\rfloor_{l_1}^{\{l_3\}} \mid \lfloor\langle t_2\rangle.Q\rfloor_{l_2}^{\{l_3\}} \mid \lfloor R\{t_1/x\} \mid !(y).R\rfloor_{l_3}^{\{l_1,l_2\}}\Big) \xrightarrow{\nu z.\langle z\rangle:\overline{l_2}\{l_3\}} \\
&\Big(\{t_1/x\}^{\{l_1,l_3\}} \mid \{t_2/z\}^{\{l_2,l_3\}} \mid \lfloor P\rfloor_{l_1}^{\{l_3\}} \mid \lfloor Q\rfloor_{l_2}^{\{l_3\}} \mid \lfloor R\{t_1/x\} \mid !(y).R\rfloor_{l_3}^{\{l_1,l_2\}}\Big) \xrightarrow{(t_2)^{\{l_2,l_3\}}:\{l_3\in\{l_2,l_3\}\}} \\
&\Big(\{t_1/x\}^{\{l_1,l_3\}} \mid \{t_2/z\}^{\{l_2,l_3\}} \mid \lfloor P\rfloor_{l_1}^{\{l_3\}} \mid \lfloor Q\rfloor_{l_2}^{\{l_3\}} \mid \lfloor R\{t_2/z\} \mid R\{t_1/x\} \mid !(y).R\rfloor_{l_3}^{\{l_1,l_2\}}\Big).
\end{aligned}
$$

Each labelled 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.

4.8.3 Example for mobility

The next example illustrates the mobility of nodes and the message loss occur when the message sent by a node N_1 has not been received by an another node, N_2, because N_2 has moved out of the transmission range of N_1. This scenario can be seen in the Figure 6. In this scenario, at first two nodes N_1 and N_2 have no connection, then, after that N_2 gets into the transmission range of the node N_1. After this the message t broadcast by N_1 is received by N_2. In the next reduction step, N_2 moves out of the transmission range of N_1, and then the message t sent by N_1 will not be intercepted by N_2, thus, t is lost.

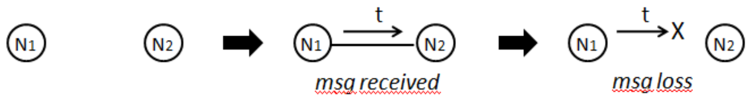


Figure 6: Example for mobility and message loss.

We let N_1 and N_2 be $\lfloor !\langle t\rangle.P_1\rfloor_{l_1}$ and $\lfloor !(x).P_2\rfloor_{l_2}$, respectively. At first, they are not connected. After the reduction relation $\rightarrow_{\{l_1\bullet l_2\}}$ the node N_2 moves into the transmission range of the node N_1. Then node N_1 broadcasts t, which is then received by N_2. These steps are modelled by the labelled transitions $\xrightarrow{\nu x.\langle x\rangle:\overline{l_1}\{l_2\}}$ and $\xrightarrow{(t)^{\{l_1,l_2\}}:\{l_2\in\{l_1,l_2\}\}}$. After these labelled transition steps, the operation of nodes N_1 and N_2 get into the states P_1 and $P_2\{t/x\}$, respectively. As the next step, node N_2 moved out of the transmission range of N_1. This is modelled by the reduction relation $\rightarrow_{\{l_1\circ l_2\}}$. At this time, node N_1 again broadcasts t, which now is not intercepted by N_2.

$$
\begin{aligned}
&\big(\lfloor !\langle t\rangle.P_1\rfloor_{l_1} \mid \lfloor !(x).P_2\rfloor_{l_2}\big) \rightarrow_{\{l_1\bullet l_2\}} \Big(\lfloor !\langle t\rangle.P_1\rfloor_{l_1}^{l_2} \mid \lfloor !(x).P_2\rfloor_{l_2}\Big) \xrightarrow{\nu x.\langle x\rangle:\overline{l_1}\{l_2\}} \\
&\Big(\{t/x\}^{\{l_1,l_2\}} \mid \lfloor P_1 \mid !\langle t\rangle.P_1\rfloor_{l_1}^{l_2} \mid \lfloor !(x).P_2\rfloor_{l_2}\Big) \xrightarrow{(t)^{\{l_1,l_2\}}:\{l_2\in\{l_1,l_2\}\}} \\
&\Big(\{t/x\}^{\{l_1,l_2\}} \mid \lfloor P_1 \mid !\langle t\rangle.P_1\rfloor_{l_1}^{l_2} \mid \lfloor P_2\{t/x\} \mid !(x).P_2\rfloor_{l_2}\Big) \rightarrow_{\{l_1\circ l_2\}} \\
&\big(\{t/x\}^{\{l_1,l_2\}} \mid \lfloor P_1 \mid !\langle t\rangle.P_1\rfloor_{l_1} \mid \lfloor P_2\{t/x\} \mid !(x).P_2\rfloor_{l_2}\big) \xrightarrow{\nu y.\langle y\rangle:\overline{l_1}\{l_2\}}
\end{aligned}
$$

$\big(\{t/y\}^{\{l_1\}} \mid \{t/x\}^{\{l_1,l_2\}} \mid \lfloor P_1 \mid !\langle t\rangle.P_1\rfloor_{l_1} \mid \lfloor P_2\{t/x\} \mid !(x).P_2\rfloor_{l_2}\big)$.

In the next subsection, we introduce the static equivalent and labeled bisimilarity in the new context, in particular, on wireless ad-hoc networks. We can use them to prove security properties of secure on-demand source routing protocols, as well as reasoning about attacks.

4.9 Attacker knowledge base, static equivalence, labeled bisimilarity

We let $\mathcal{L}(N)$ be the set of identifier l_i's in the network N, again we recall that each l_i is a unique name in the network, and identify each node of the network. Then we 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 follow:

$$E{=}\nu\tilde{n}.\,(\{^{t_1}/_{x_1}\}^{\sigma_1} \mid \{^{t_2}/_{x_2}\}^{\sigma_2} \mid \ldots \mid \{^{t_m}/_{x_m}\}^{\sigma_m} \mid N_1 \mid \ldots \mid N_r)$$

The output of the extended network E is defined by a *frame* φ, which is composed of name restrictions and a parallel composition of **all** active substitutions:
$\varphi = \nu\tilde{n}.\,(\{^{t_1}/_{x_1}\} \mid \{^{t_2}/_{x_2}\} \mid \ldots \mid \{^{t_m}/_{x_m}\})$. We note that in φ the ranges $\sigma_1, \ldots, \sigma_n$ are removed from active substitutions with range.

Intuitively, the frame represents the output of the network. We note that while the attacker node knows only the messages sent by its neighbors the wireless environment knows all of the sent messages. By hearing everything the wireless environment can distinguish the operation of two networks. When an attack (against a routing protocol) is executed successfuly on a specific topology the wireless environment will be aware of it, because it can distinguish the correct 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 messages from its neighbors. On the other hand, the attacker can perform numerous computation steps based on its knowledge. Adapting the notion of frame, 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 φ, because it contains only such active substitution(s) $\{^{t_i}/_{x_i}\}^{\sigma_i}$ $(i{\in}\{1,\ldots,m\})$ where $l_a \in \sigma_i$. That is,

$$\varphi(l_a){=}\nu\tilde{n}.\,\big(\{^{t_i}/_{x_i}\}^{\sigma_i} \mid \{^{t_j}/_{x_j}\}^{\sigma_j} \mid \ldots \mid \{^{t_k}/_{x_k}\}^{\sigma_k}\big),$$

where $l_a \in \sigma_i$, $l_a \in \sigma_j$,..., $l_a \in \sigma_k$, and $\{i, j, \ldots, k\} \subseteq \{1, 2 \ldots, n\}$.

Definition 9. *Two terms t_1 and t_2 are equal in a frame φ, and write $[t_1 = t_2]\,\varphi$, if and only if $\varphi \equiv \nu\widetilde{n}.\omega$, $t_1\omega = t_2\omega$, and $\{\widetilde{n}\} \cap (fn(t_1) \cup fn(t_2)) = \emptyset$ for some names $\widetilde{n}$ and substitution ω.*

Definition 10. *Two closed frames φ and ψ are statically equivalent, and write $\varphi \approx_s \psi$, when $dom(\varphi) = dom(\psi)$ 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.

Lemma 1. *Static equivalence is closed by structural equivalence, by reduction, and by application of closing evaluation contexts C[-].*

Proof. Due to the definition of frame we use in static equivalent is the same as in the applied π-calculus. The proof is also the same as in [10]. □

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 equivalent it is enough to verify the frames we already know.

Unlike the spi-calculus [1], which is designed for reasoning about security protocols, where one has to define the relation $\Re$ that pairs two processes between which he wants to prove observational equivalence. Furthermore, one has to define the cipher environment and takes care about the attacker's (process R that comes into contact with the two processes as a parallel composition) condition after each relation step, that is, one has to prove that secret key materials still not be obtained by the attacker (that is, names represents secret keys not become a free name of R).

Finally, we define the labeled bisimilarity in the context of Mobile Ad-Hoc Network. The advantage of the labeled bisimilarity is that it does not depend on an arbitrary context but only on the frames which is well-known after each transition step.

In order to make the definition be intuitive in the context of Mobile Ad-Hoc Networks, in the next definition without corrupting the correctness we assume that

E_1 consists of one plain network N_1, and E_2 consists of one plain network N_2. Again, we note that we are considering only the reasoning about the attacks discussed in Section 2, hence, we assume that the topology remains unchanged during attacks

Definition 11. *Labeled bisimilarity ($\approx_l^N$) is the largest symmetric relation $\Re$ on closed extended networks such that $E_1 \Re E_2$ implies: $\mathcal{L}(N_1) = \mathcal{L}(N_2)$ and $connect_i(\mathcal{L}(N_1)) = connect_j(\mathcal{L}(N_2))$, and*

1. *$E_1 \approx_s E_2$;*
2. *if $E_1 \longrightarrow E_1'$, then $E_2 \longrightarrow^* E_2'$ and $E_1' \Re E_2'$ for some E_2'; (This is the induction based on internal reductions);*
3. *if $E_1 \xrightarrow{\alpha} E_1'$ and $fv(\alpha) \subseteq dom(E_1)$ and $bn(\alpha) \cap fn(E_2) = \emptyset$; then $E_2 \longrightarrow^* \xrightarrow{\alpha} \longrightarrow^* E_2'$ and $E_1' \Re E_2'$ for some E_2'. (This is the induction based on labeled relations). Here α can be a broadcast, an unicast, or a receive action.*

Intuitively, this definition means that the outputs of the two networks of same topology cannot be distinguished 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, $\rightarrow^*$ models the sequential execution of some internal reductions, or more formally, a transitive and reflexive closure of $\rightarrow$.

Definition 12. *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 in every same topology. That is, $\forall connect_i \in \mathcal{C}(N_1)$, $\forall connect_j \in \mathcal{C}(N_2)$, such that $connect_i(\mathcal{L}(N_1)) = connect_j(\mathcal{L}(N_2))$: $E_1 \approx_l^N E_2$.*

4.10 Example on modelling the attacker knowledge base

Let us consider the example topologies in the Figure 7. Next we demonstrate how to model that the attacker node collects information, namely, how the attacker builds its knowledge base during the route discovery phase.

First, we consider the scenario on the left side. Let N_1 be $\lfloor P_1 \rfloor_{l_1}^{\{l_2,l_a\}}$, N_2 be $\lfloor P_2 \rfloor_{l_2}^{\{l_1,l_a\}}$, N_3 be $\lfloor P_3 \rfloor_{l_3}^{\{l_a,l_4\}}$, N_4 be $\lfloor P_4 \rfloor_{l_4}^{\{l_a,l_3\}}$, and N_A be $\lfloor P_A \rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}}$. Then the topology on the left of the Figure 7 is specified as:

$$netw_1 \overset{def}{=} \lfloor P_1 \rfloor_{l_1}^{\{l_2,l_a\}} \mid \lfloor P_2 \rfloor_{l_2}^{\{l_1,l_a\}} \mid \lfloor P_A \rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \mid \lfloor P_3 \rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4 \rfloor_{l_4}^{\{l_a,l_3\}}$$

After the broadcasting of the request req_1 by node N_1 $netw_1$ gets into the state $netw_1'$.

$$netw_1 \overset{\nu x.\langle x\rangle:\overline{l_1}\{l_2,l_a\}}{\longrightarrow} netw_1', \text{ where}$$

$$netw_1' \overset{def}{=} \{^{req_1}/_x\}^{\{l_2,l_a\}} \mid \lfloor P_1' \rfloor_{l_1}^{\{l_2,l_a\}} \mid \lfloor P_2 \rfloor_{l_2}^{\{l_1,l_a\}} \mid \lfloor P_A \rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \mid \lfloor P_3 \rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4 \rfloor_{l_4}^{\{l_a,l_3\}}$$

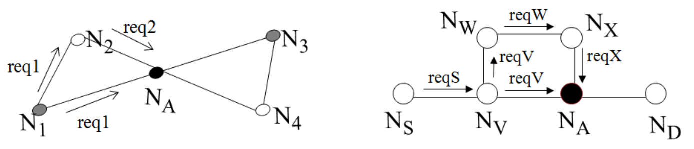


Figure 7: On the left is the topology in which an attack is found against the SRP protocol: Node N_1 initiates the route discovery towards node N_3, and node N_A is the attacker node. On the right is the topology in which an attack is found against the Ariadne protocol: Node N_S initiates the route discovery towards node N_D, and node N_A is the attacker node.

The active substitution with range $\{^{req_1}/_x\}^{\{l_2,l_a\}}$ means that the attacker node intercepts the request req_1 because $l_a \in \{l_2, l_a\}$. Thus, at this time the knowledge base of the attacker node is increased with req_1. The process P_1' is the process we reach from P_1 after broadcasting req_1. After N_2 broadcasts req_2, the network $netw_1'$ reaches the state $netw_1''$:

$$netw_1' \xrightarrow{\nu y.\langle y\rangle:\overline{l_2}\{l_1,l_a\}} netw_1'', \text{ where}$$

$$netw_1'' \stackrel{def}{=} \{^{req_2}/_y\}^{\{l_1,l_a\}} \mid \{^{req_1}/_x\}^{\{l_2,l_a\}} \mid \lfloor P_1' \rfloor_{l_1}^{\{l_2,l_a\}} \mid \lfloor P_2' \rfloor_{l_2}^{\{l_1,l_a\}} \mid \lfloor P_A \rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \mid \\ \lfloor P_3 \rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4 \rfloor_{l_4}^{\{l_a,l_3\}}$$

The active substitution with range $\{^{req_2}/_y\}^{\{l_1,l_a\}}$ means that the attacker node intercepts the request req_2 because $l_a \in \{l_1, l_a\}$. Hence, at this point the knowledge of the attacker node has been extended with req_1 and req_2. Formally, let us assume that the initial knowledge of the attacker is zero, then we have $\varphi(l_a) = \{^{req_2}/_y\} \mid \{^{req_1}/_x\}$.

The scenario on the right side in the Figure 7 can be described in the same manner as in the scenario on the left side.

We note that this ability of the sr-calculus is novel compared to CMAN and the ω-calculus. With this ability the sr-calculus can be used to directly modelling the attacks found against the SRP and Ariadne protocols, which is not the case in CMAN and the ω-calculus.

4.11 Example on labeled bisimilarity ($\approx_l^N$)

Let us consider the example network and scenario $netw_1$ in the Figure 8 that is similar to the case in the Figure 5. In addition, let us consider the another network $netw_2$, which has the same topology (node identifiers and neighborhood) as $netw_1$. However, in $netw_2$ the first node has the internal operation $\langle t_3\rangle.P'$ in which the message t_3 is broadcast instead of t_1 as in the case of $\langle t_1\rangle.P$, where $t_1 \neq t_3$.

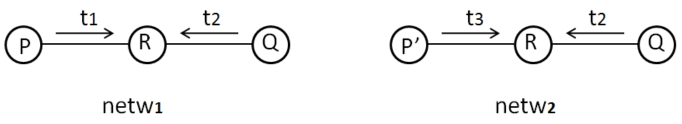


Figure 8: The two networks $netw_1$ and $netw_2$ have the same topology but the operations of the leftmost nodes differ (P and P').

$$netw_1 \stackrel{def}{=} \left(\lfloor \langle t_1\rangle.P \rfloor_{l_1}^{\{l_3\}} \mid \lfloor \langle t_2\rangle.Q \rfloor_{l_2}^{\{l_3\}} \mid \lfloor !(y).R \rfloor_{l_3}^{\{l_1,l_2\}}\right) \xrightarrow{\nu x.\langle x\rangle:\overline{l_1}\{l_3\}}$$

$$\left(\{t_1/x\}^{\{l_3\}} \mid \lfloor P \rfloor_{l_1}^{\{l_3\}} \mid \lfloor \langle t_2\rangle.Q \rfloor_{l_2}^{\{l_3\}} \mid \lfloor !(y).R \rfloor_{l_3}^{\{l_1,l_2\}}\right) \xrightarrow{(t_1)^{\{l_3\}}:\{l_2\in\{l_3\}\}}$$

$$\Big(\{t_1/x\}^{\{l_3\}} | \lfloor P \rfloor_{l_1}^{\{l_3\}} \mid \lfloor \langle t_2 \rangle.Q \rfloor_{l_2}^{\{l_3\}} \mid \lfloor R\{t_1/x\} \mid !(y).R \rfloor_{l_3}^{\{l_1,l_2\}}\Big) \xrightarrow{\nu z.\langle z \rangle : \overline{l_2}\{l_3\}}$$
$$\Big(\{t_1/x\}^{\{l_3\}} | \{t_2/z\}^{\{l_3\}} | \lfloor P \rfloor_{l_1}^{\{l_3\}} \mid \lfloor Q \rfloor_{l_2}^{\{l_3\}} \mid \lfloor R\{t_1/x\} \mid !(y).R \rfloor_{l_3}^{\{l_1,l_2\}}\Big) \xrightarrow{(t_2)^{\{l_3\}} : \{l_2 \in \{l_3\}\}}$$
$$\Big(\{t_1/x\}^{\{l_3\}} | \{t_2/z\}^{\{l_3\}} | \lfloor P \rfloor_{l_1}^{\{l_3\}} \mid \lfloor Q \rfloor_{l_2}^{\{l_3\}} \mid \lfloor R\{t_2/z\} | \; R\{t_1/x\} \mid !(y).R \rfloor_{l_3}^{\{l_1,l_2\}}\Big).$$

$$netw_2 \overset{def}{=} \Big(\lfloor \langle t_3 \rangle.P' \rfloor_{l_1}^{\{l_3\}} \mid \lfloor \langle t_2 \rangle.Q \rfloor_{l_2}^{\{l_3\}} \mid \lfloor !(y).R \rfloor_{l_3}^{\{l_1,l_2\}}\Big) \xrightarrow{\nu x.\langle x \rangle : \overline{l_1}\{l_3\}}$$
$$\Big(\{t_3/x\}^{\{l_3\}} | \lfloor P' \rfloor_{l_1}^{\{l_3\}} \mid \lfloor \langle t_2 \rangle.Q \rfloor_{l_2}^{\{l_3\}} \mid \lfloor !(y).R \rfloor_{l_3}^{\{l_1,l_2\}}\Big) \xrightarrow{(t_3)^{\{l_3\}} : \{l_2 \in \{l_3\}\}}$$
$$\Big(\{t_3/x\}^{\{l_3\}} | \lfloor P' \rfloor_{l_1}^{\{l_3\}} \mid \lfloor \langle t_2 \rangle.Q \rfloor_{l_2}^{\{l_3\}} \mid \lfloor R\{t_1/x\} \mid !(y).R \rfloor_{l_3}^{\{l_1,l_2\}}\Big) \xrightarrow{\nu z.\langle z \rangle : \overline{l_2}\{l_3\}}$$
$$\Big(\{t_3/x\}^{\{l_3\}} | \{t_2/z\}^{\{l_3\}} | \lfloor P' \rfloor_{l_1}^{\{l_3\}} \mid \lfloor Q \rfloor_{l_2}^{\{l_3\}} \mid \lfloor R\{t_1/x\} \mid !(y).R \rfloor_{l_3}^{\{l_1,l_2\}}\Big) \xrightarrow{(t_2)^{\{l_3\}} : \{l_2 \in \{l_3\}\}}$$
$$\Big(\{t_3/x\}^{\{l_3\}} | \{t_2/z\}^{\{l_3\}} | \lfloor P' \rfloor_{l_1}^{\{l_3\}} \mid \lfloor Q \rfloor_{l_2}^{\{l_3\}} \mid \lfloor R\{t_2/z\} | \; R\{t_1/x\} \mid !(y).R \rfloor_{l_3}^{\{l_1,l_2\}}\Big).$$

It is easy to see that $netw_1$ and $netw_2$ are **not** labeled bisimilar, because their outputs, that is their frames $\{t_1/x\}|\{t_2/z\}$ and $\{t_3/x\}|\{t_2/z\}$ can be distinguished because of $t_1 \neq t_3$.

5 Attacker's ability and knowledge

The computation ability of the attacker is an unchanged set, denoted by $\mathcal{A}_c$, of constructor functions such as computing encryption, hash, and digital signature, compose a message tuple etc. The knowledge of the attacker is composed of initial knowledge (denoted by $\mathcal{K}_{init}$) and gained knowledge (denoted by $\mathcal{K}_{gain}$). Typically, $\mathcal{K}_{init}$ often contains the node IDs of the neighborhood of the attacker, and pre-shared keys. Hence, formally, when modeling the initial knowledge we initiate the frame $\varphi(l_a)$ with the substitution (with the range $\{l_a\}$, hence, not available for the honest nodes) of the initial knowledge on new variables. Thereafter, the frame $\varphi(l_a)$ is periodically extended with new knowledge (i.e., $\mathcal{K}_{gain}$), and whenever the attacker compute a message for his purpose it can use its whole knowledge and computation ability.

The computation ability of the attacker node is the set $\mathcal{B}$ 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 $\mathcal{B}$ also contains equations from $\sum$, such as *dec(enc(x, y),y)* $=$ *x* and *checksign(sign(x, y),pk(y))* $=$ *true* for a special constant *true.* In the processes of the attacker, the parameters of these functions and equations can only those that appear in $\varphi(l_a)$.

To make the behavior of the attacker systematic, we assume that the attacker tries all the possible moves (selecting possible functions, equations with the available parameters). To reduce the number of possibilities we can explicitly add the type-respect binding of the parameters to functions and equations. For instance, in *sign(t,k)*, from $\varphi(l_a)$ only the terms of type DATA can be bound to t, and only terms of type PRIVATEKEY can be bound to k. Moreover, in case of source routing protocols, usually, the patterns (skeletons) of the accepted reply and request are known. Hence, by reasoning in a backward manner that which kind of message parts the attacker need to have in order to compose the reply or request that includes an invalid route but fullfils the pattern of accepted message, we can systematically make the analysis and greatly reducing the number of possibilities.

6 Application of the calculus

In this section we demonstrate the usability of the *sr*-calculus by modelling the SRP protocol and the attack scenario we discussed in Section 2.

In order to model secure on-demand source routing protocols for mobile ad-hoc networks we introduce the following required constructor functions and destructor applications.

We start with the discussion of the construction function tuple and the next and previous functions related to tuple, then we discuss the MAC function.
tuple: The constructor function **tuple** models a tuple of n terms t_1, t_2,..., t_n. We write the function as

$$tuple(t_1, t_2, \ldots, t_n)$$

We abbreviate it simply as $(t_1, t_2, \ldots, t_n)$ in the rest of the paper.

We introduce the destructor functions **i** that returns the i-th element of a tuple of n elements, where $i \in \{1, \ldots, n\}$:

$$i(t_1, t_2, \ldots, t_n) = t_i$$

list: The constructor function **list** models a list of n terms t_1, t_2,..., t_n. We write the function as

$$list(t_1, t_2, \ldots, t_n)$$

We abbreviate it as $[t_1, t_2, \ldots, t_n]$ in the rest of the paper. We note that a list can be empty, that is $n = 0$, and we denote the empty list as [ ].

Then the destructor applications **next** and **prev** are introduced for modelling the next and the previous element of a particular element in the list. Each function has two arguments, a first is a *list* and the second is the term of which we want to know its next and previous element in the given list. If there is no such element in the list or it has no next or previous element then it returns a constant symbol ***undefined***. The sort system may enforce that *next* and *prev* are applied only to list.

$$\begin{aligned}
&next([t_1, \ldots, t_i, t_{i+1}, \ldots t_n], t_i) = t_{i+1},\\
&next([t_1, \ldots, t_n], t_n) = undefined\\
&next([t_1, \ldots, t_n], t_i) = undefined, \text{ if } t_i \text{ does not occur in the list,}\\
&prev([t_1, \ldots, t_{i-1}, t_i, \ldots t_n], t_i) = t_{i-1},\\
&prev([t_1, \ldots, t_n], t_1) = undefined\\
&prev([t_1, \ldots, t_n], t_i) = undefined, \text{ if } t_i \text{ does not occur in the list.}
\end{aligned}$$

We also introduce the functions **toendlist** and **toheadlist** that model the list with $n+1$ elements by appending an element t to the end (to the head) of a list t_{list} of n elements with same sort as t, respectively.

$$\begin{aligned}
&toendlist\,([t_1, \ldots, t_n], t) = [t_1, \ldots, t_n, t]\\
&toheadlist\,(t, [t_1, \ldots, t_n]) = [t, t_1, \ldots, t_n].
\end{aligned}$$

In the rest of the paper for convenient presentation we write $[t_1, \ldots, t_n, t]$ instead of $toendlist\,([t_1, \ldots, t_n], t)$. With the function *toendlist* we can model lists as follows: $[l_1, l_2, \ldots, l_n] = toendlist(\ldots toendlist(toendlist([\,], l_1), l_2) \ldots l_n)$.

Functions *first*, and *last* represents the first, and the last element of *List*, respectively.

$$\begin{aligned}
&first([t_1, \ldots, t_n]) = t_1\\
&last([t_1, \ldots, t_n]) = t_n.
\end{aligned}$$

Finally we model the keyed hash or MAC function with symmetric key k with the binary function **mac**. The

$$mac(t_1, t_2).$$

function that computes the message authentication code of message t_1 using secret key t_2. The shared key between node l_i and l_j is modelled by function $k(l_i, l_j)$.

6.1 Modelling the SRP protocol and the attack

The scenario in Section 2 is modelled by the extended network defined as:

$$netw \overset{def}{=} (\lfloor P_1 \rfloor_{l_1}^{\{l_2,l_a\}} \mid \lfloor P_2 \rfloor_{l_2}^{\{l_1,l_a\}} \mid \lfloor !P_A \rfloor_{l_a}^{\{l_1,l_2 l_3,l_4\}} \mid \lfloor !P_3 \rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4 \rfloor_{l_4}^{\{l_a,l_3\}}).$$

where $N_1 {=} \lfloor P_1 \rfloor_{l_1}^{\{l_2,l_a\}}$, $N_2 {=} \lfloor P_2 \rfloor_{l_2}^{\{l_1,l_a\}}$, $A {=} \lfloor !P_A \rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}}$, $N_3 {=} \lfloor !P_3 \rfloor_{l_3}^{\{l_a,l_4\}}$, $N_4 {=} \lfloor P_4 \rfloor_{l_4}^{l_a l_3}$. Processes P_1,P_2,$!P_3$,P_4 model the operation of honest nodes while process A model the operation of the attacker node as follows:

$$\begin{aligned}
P_1 &\overset{def}{=} \mathit{let}\ MAC_{13} = mac\,((l_1,l_3),k(l_1,l_3))\ \ \mathit{in}\ ReqInit.\\
ReqInit &\overset{def}{=} \langle (req, l_1, l_3, MAC_{13}, [\,]) \rangle .!\,WaitRep_1.\\
WaitRep_1 &\overset{def}{=} (x_{rep}).[1(x_{rep}) = l_1]\ [2(x_{rep}) = rep]\ [3(x_{rep}) = l_1]\\
&\quad [4(x_{rep}) = l_3][first(5(x_{rep})) \in \{l_2 l_a\}]\\
&\quad [mac\,((l_1,l_3,5(x_{rep}))\,,k(l_1,l_3)) = 6(x_{rep})]\\
&\quad \langle ACCEPT \rangle.
\end{aligned}$$

Intuitively, the node l_1 generates the route request message that includes the ID of source and target nodes, and the message authentication code MAC_{13} computed using the shared key, broadcasts it and waits for the reply. When it receives a message, it checks whether (i) it is the addressee, (ii) the message is a reply, (iii) the ID of the source and the target nodes, and (iv) the message authentication code using its shared key. If all are correct then it signals term $ACCEPT$. The process P_2 models the operation of the node N_2 and is specified as follow:

$$\begin{aligned}
P_2 &\overset{def}{=} (y_{req})\,.[1(y_{req}) = req].\\
&\quad \langle (1(y_{req}), 2(y_{req}), 3(y_{req}), 4(y_{req}), [5(y_{req}), l_2]) \rangle\\
&\quad !\,WaitRep_2.\\
WaitRep_2 &\overset{def}{=} (y_{rep})\,.[1(y_{rep}) = l_2][2(y_{rep}) = rep]\\
&\quad [next\,(5(y_{rep}), l_2) \in \{l_1 l_a\}]\\
&\quad \mathit{in}\ \langle (l_1, 2(y_{rep}), 3(y_{rep}), 4(y_{rep}), 5(y_{rep}), 6(y_{rep})) \rangle.
\end{aligned}$$

Intuitively, on receiving a message it checks if it is a request, then appends its ID l_2 to the end of the list, re-broadcasts it and waits for a reply. When it receives the reply message it checks if the message is intended to it, it is a reply, the next ID in the list corresponds to neighbors and forwards the message to the destination node l_1. The process P_4 models the operation of the node N_4 and is specified as follow:

$$\begin{aligned}
P_4 &\overset{def}{=} (z_{req})\,.[1(z_{req}) = req].\\
&\quad \langle (1(z_{req}), 2(z_{req}), 3(z_{req}), 4(z_{req}), [5(z_{req}), l_4]) \rangle\\
&\quad !\,WaitRep_4.\\
WaitRep_4 &\overset{def}{=} (z_{rep})\,.[1(z_{rep}) = l_2][2(z_{rep}) = rep]\\
&\quad [prev\,(5(z_{rep}, l_4)) \in \{l_a l_3\}]\\
&\quad \mathit{let}\ t_{idList} = 5(z_{rep})\ \ \mathit{in}\ \mathit{let}\ l_{prev} = prev\,(t_{idList}, l_4)\\
&\quad \mathit{in}\ \langle (l_{prev}, 2(z_{rep}), 3(z_{rep}), 4(z_{rep}), 5(z_{rep}), 6(z_{rep})) \rangle.
\end{aligned}$$

Intuitively, on receiving a message node N_4 checks if it is a request, then appends its identifier l_4 to the end of the list, re-broadcasts it and waits for a reply. When it receives the reply message it checks if the message is intended to it, it is a reply, the previous and next ID in the list corresponds to neighbors and forwards the message to the previous node l_{prev} in the list. The process P_4 models the operation of the node N_4 and is specified as follow:

Finally, the operation of the destination node N_3, the process P_3, is modelled as:

$$\begin{aligned}
P_3 &\overset{def}{=} (w_{req})\,.[1(w_{req}) = req][3(w_{req}) = l_3].\\
&\quad [mac\,(\langle 2(w_{req}), 3(w_{req}) \rangle, k(l_1,l_3)) = 4(w_{req})]\ \ \mathit{let}\ MAC_{31} =
\end{aligned}$$

$$\begin{aligned}&mac((1(w_{req}),2(w_{req}),3(w_{req}),5(w_{req}))\,,k(l_1,l_3))\ \ in\\ &let\ l_{prev}=last\,(5(w_{req}))\ \ in\\ &\langle(l_{prev},rep,2(w_{req}),3(w_{req}),5(w_{req}),MAC_{31})\rangle.\end{aligned}$$

Intuitively, on receiving the a message it checks if the message is a request, and it is the destination, and verifies the MAC embedded in the request using its shared key with l_1. If so then it creates a reply message and forwards it to the last node in the list.

Next we specify the model ($\mathcal{M}_A$) of the attacker node as follows: we assume that the attacker cannot forge message authentication codes MAC_{13} and MAC_{31} without possessing keys. Initially, the attacker node knows the IDs of its neighbors $\{l_1,l_2,l_3,l_4\}$. The attacker can creates new data n, and can append elements of $\{l_1,l_2,l_3,l_4\}$, and n to the end of an ID list it receives. Finally, it can broadcast and unicast its message to honest nodes.

The attacker overhears only messages sent by its neighbors. Let frame $\varphi(l_a)$ be $\{^{t_i}/_{x_i}\}\ |\ \{^{t_j}/_{x_j}\}\ |\ \ldots\ |\ \{^{t_k}/_{x_k}\}$. This represents the attacker's knowledge he accumulates during the route discovery phase by eavesdropping. He combines this accumulated knowledge and initial knowledge to construct an attack. Let T_{l^p} 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: $P_A \overset{def}{=} (\tilde{x})\,.\nu n.\langle f\,(\tilde{x},T_{l^p},n)\rangle$, where $\tilde{x}$ is a tuple $(x_1,\ldots,x_n)$ of variables, νn means the attacker creates new data n. The function $f\,(\tilde{x},T_{l^p},n)$ represents the message the attacker generates from the eavesdropped messages that it receives by binding them to $\tilde{x}$, its initial knowledge and the newly generated data n, respectively. At first, $\tilde{x}$ is a single variable x_a.

As the next step, we define an ideal model of *netw*, written as $netw_{spec}$. The definition of $netw_{spec}$ is the same as *netw* except that the desription of N_1 is $\lfloor P_1^{spec}\rfloor_{l_1}^{\{l_2 l_a\}}$.

Process P_1^{spec} 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 a special function $consistent(List)$ that informs it about the correctness of the returned route. We define this ideal source node as follow:

$$\begin{aligned}P_1^{spec} &\overset{def}{=} let\ MAC_{13}=mac\,((l_1,l_3),k(l_1,l_3))\ \ in\ ReqInit_{spec}.\\ ReqInit_{spec} &\overset{def}{=} \langle(req,l_1,l_3,MAC_{13},[\,])\rangle.!\,WaitRep_{spec}.\\ WaitRep_{spec} &\overset{def}{=} (x_{rep}).[1(x_{rep})=l_1]\ [2(x_{rep})=rep][3(x_{rep})=l_1]\\ &\quad [4(x_{rep})=l_3][first(5(x_{rep}))\in\{l_2l_a\}]\\ &\quad [mac\,((l_1,l_3,5(x_{rep}))\,,k(l_1,l_3))=6(x_{rep})]\\ &\quad [consistent\,(5(x_{rep}))=true].\langle ACCEPT\rangle.\end{aligned}$$

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. Next we give the *definition of secure routing* based on labeled bisimilarity:

Definition 13. *A routing protocol is said to be secure if for all extended networks E and its corresponding ideal network E_{spec}, which includes an arbitrary attacker node, we have: $E \approx_l^N E_{spec}$.*

Theorem 1. *The SRP protocol is insecure.*

Proof. We will show that $netw \approx_l^N netw_{spec}$ does **not** hold besides the attacker $\mathcal{M}_A$ because the third point of the Definition 11 is violated. In order to do this we will show that there exist a sequence of labelled transitions and internal reduction relations that can be performed in case of *netw* but can not be performed in case of $netw_{spec}$. Formally, this means that the frames of *netw* and $netw_{spec}$ can be distinguished.

Let us see the following sequence of labelled transitions and reduction relations that *netw* can perform: First the source node l_1 broadcast the route request message $(req,l_1,l_3,MAC_{13},[\,])$ to initiates the discovery of the route towards the node l_3.

$\Big(\lfloor P_1 \rfloor_{l_1}^{\{l_2,l_a\}} \mid \lfloor P_2 \rfloor_{l_2}^{\{l_1,l_a\}} \mid \lfloor !P_A \rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \mid \lfloor !P_3 \rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4 \rfloor_{l_4}^{\{l_a,l_3\}}\Big) \xrightarrow{\nu x.\langle x\rangle : \overline{l_1}\{l_2,l_a\}}$

$(\{^{(req,l_1,l_3,MAC_{13},[\,])}/_x\}^{\{l_1,l_2,l_a\}} \mid \lfloor P_1' \rfloor_{l_1}^{\{l_2,l_a\}} \mid \lfloor P_2 \rfloor_{l_2}^{\{l_1,l_a\}} \mid \lfloor !P_A \rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \mid \lfloor !P_3 \rfloor_{l_3}^{\{l_a,l_4\}}$
$\quad \mid \lfloor P_4 \rfloor_{l_4}^{\{l_a,l_3\}})$ ($\to \times 2$ (EXT BroadRecv1))

The active substitution with range $\{^{(req,l_1,l_3,MAC_{13},[\,])}/_x\}^{\{l_1,l_2,l_a\}}$ means that after l_1 broadcasts the message $(req, l_1, l_3, MAC_{13}, [\,])$ it is available for itself and its neighbors, the nodes l_1 and l_a. The process P_1' is $!\,WaitRep_1$. After applying two times the rule (EXT BroadRecv1) *netw* reaches the following state:

$(\{^{(req,l_1,l_3,MAC_{13},[\,])}/_x\}^{\{l_1,l_2,l_a\}} \mid \lfloor !\,WaitRep_1 \rfloor_{l_1}^{\{l_2,l_a\}}$
$\quad \mid \lfloor P_2 \{^{(req,l_1,l_3,MAC_{13},[\,])}/_x\} \rfloor_{l_2}^{\{l_1,l_a\}}$
$\quad \mid \lfloor P_A \{^{(req,l_1,l_3,MAC_{13},[\,])}/_x\} \mid !P_A \rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}}$
$\quad \mid \lfloor !P_3 \rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4 \rfloor_{l_4}^{\{l_a,l_3\}})$ ($\to \times 2$ (Red P-Let))

Intuitively, this means that node l_2 and the attacker node l_a receive the broadcast message. After broadcasting the message $(req, l_1, l_3, MAC_{13}, [\,])$ node l_1 reaches to the state $!\,WaitRep_1$ and after receiving the route request message the nodes l_2 and l_a are going to broadcast their message. The node l_2 is going to broadcast the request message $(req, l_1, l_3, MAC_{13}, [l_2])$ and the attacker node $(req, l_1, l_3, MAC_{13}, [l_2, n, l_4])$. ($\to \times 2$ (Red P-Let)) means the application of the reduction relation rule (Red P-Let) twice.

$(\{^{(req,l_1,l_3,MAC_{13},[\,])}/_x\}^{\{l_1,l_2,l_a\}} \mid \lfloor !\,WaitRep_1 \rfloor_{l_1}^{\{l_2,l_a\}}$
$\quad \mid \lfloor \langle 1(x), 2(x), 3(x), 4(x), [5(x), l_2] \rangle .!\,WaitRep_2 \rfloor_{l_2}^{\{l_1,l_a\}}$
$\quad \mid \lfloor \langle 1(x), 2(x), 3(x), 4(x), [l_2, n, l_4] \rangle \mid !P_A \rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}}$
$\quad \mid \lfloor !P_3 \rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4 \rfloor_{l_4}^{\{l_a,l_3\}}) \xrightarrow{\nu y.\langle y\rangle : \overline{l_a}\{l_1,l_2,l_3,l_4\}}$

As we mentioned earlier the broadcast sends are choosed non-deterministically they are going to executed at the same time. We assume that in this labelled transition trace the attacker node outputs its message before node l_2. After broadcasting $(req, l_1, l_3, MAC_{13}, [l_2, n, l_4])$ *netw* reaches the following state:

$(\{^{(req,l_1,l_3,MAC_{13},[\,])}/_x\}^{\{l_1,l_2,l_a\}} \mid \{^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_y\}^{\{l_a,l_1,l_2,l_3,l_4\}}$
$\quad \mid \lfloor !\,WaitRep_1 \rfloor_{l_1}^{\{l_2,l_a\}} \mid \lfloor \langle 1(x), 2(x), 3(x), 4(x), [5(x), l_2] \rangle .!\,WaitRep_2 \rfloor_{l_2}^{\{l_1,l_a\}}$
$\quad \mid \lfloor 0 \mid !P_A \rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \mid \lfloor !P_3 \rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4 \rfloor_{l_4}^{\{l_a,l_3\}}) \equiv_{Struct\ P-Par1}$
$\quad$ ($\to \times 4$ (EXT BroadRecv1))

We note that at this time the frame of *netw* is

$\{^{(req,l_1,l_3,MAC_{13},[\,])}/_x\}^{\{l_1,l_2,l_a\}} \mid \{^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_y\}^{\{l_a,l_1,l_2,l_3,l_4\}}$, which

represents the output messages so far. After applying the rules (Struct P-Par1) and 4 times the rule (EXT BroadRecv1) *netw* reaches the following state:

$(\{^{(req,l_1,l_3,MAC_{13},[\,])}/_x\}^{\{l_1,l_2,l_a\}} \mid \{^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_y\}^{\{l_a,l_1,l_2,l_3,l_4\}}$
$\quad \mid \lfloor WaitRep_1 \{^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_y\} \mid !\,WaitRep_1 \rfloor_{l_1}^{\{l_2,l_a\}}$
$\quad \mid \lfloor P_2' \{^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_y\} \rfloor_{l_2}^{\{l_1,l_a\}} \mid \lfloor !P_A \rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \mid \lfloor !P_3 \rfloor_{l_3}^{\{l_a,l_4\}}$
$\quad \mid \lfloor P_3 \{^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_y\} \rfloor_{l_3}^{\{l_a,l_4\}}$

$$\mid \left\lfloor P_4 \left\{{}^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_{y}\right\}\right\rfloor^{\{l_a,l_3\}}_{l_4}) \rightarrow^*$$

Here the process P_2' is $\langle 1(x), 2(x), 3(x), 4(x), [5(x), l_2]\rangle.!\,WaitRep_2$. After receiving the message $(req, l_1, l_3, MAC_{13}, [l_2, n, l_4])$ broadcast by the attacker node, nodes l_1 and l_2 drops it since the verification they make on it fails. This is modelled by the nil process.

$$\begin{aligned}
&(\left\{{}^{(req,l_1,l_3,MAC_{13},[\,])}/_{x}\right\}^{\{l_1,l_2,l_a\}} \mid \left\{{}^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_{y}\right\}^{\{l_a,l_1,l_2,l_3,l_4\}}\\
&\quad \mid \lfloor 0 \mid !\,WaitRep_1\rfloor^{\{l_2,l_a\}}_{l_1}\\
&\quad \mid \lfloor P_2'\rfloor^{\{l_1,l_a\}}_{l_2} \mid \lfloor !P_A\rfloor^{\{l_1,l_2,l_3,l_4\}}_{l_a} \mid \lfloor \langle l_4, rep, 2(y), 3(y), 5(y), MAC_{31}\rangle \mid !P_3\rfloor^{\{l_a,l_4\}}_{l_3}\\
&\quad \mid \lfloor \langle 1(y), 2(y), 3(y), 4(y), [5(y), l_4]\rangle.!\,WaitRep_4\rfloor^{\{l_a,l_3\}}_{l_4})\\
&\quad \longrightarrow^*,\ \equiv_{Struct\ P-Par1} \times 2,\ \xrightarrow{\nu x'.\langle x'\rangle:\overline{l_2}\{l_1,l_a\}}
\end{aligned}$$

After applying a sequence of reduction relations that models the verification made on the message and applying the rule (Struct P-Par1) twice the nil processes are eliminated, and applying the labeled transition $\xrightarrow{\nu x'.\langle x'\rangle:\overline{l_2}\{l_1 l_a\}}$, we have that *netw* reaches the following state:

$$\begin{aligned}
&(\left\{{}^{(req,l_1,l_3,MAC_{13},[\,])}/_{x}\right\}^{\{l_1,l_2,l_a\}} \mid \left\{{}^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_{y}\right\}^{\{l_a,l_1,l_2,l_3,l_4\}}\\
&\quad \mid \left\{{}^{(req,l_1,l_3,MAC_{13},[l_2])}/_{x'}\right\}^{\{l_2,l_1,l_a\}} \mid \lfloor !\,WaitRep_1\rfloor^{\{l_2,l_a\}}_{l_1} \mid \lfloor !\,WaitRep_2\rfloor^{\{l_1,l_a\}}_{l_2}\\
&\quad \mid \lfloor !P_A\rfloor^{l_1 l_2 l_3 l_4}_{l_a} \mid \lfloor \langle l_4, rep, 2(y), 3(y), 5(y), MAC_{31}\rangle \mid !P_3\rfloor^{\{l_a,l_4\}}_{l_3}\\
&\quad \mid \lfloor \langle 1(y), 2(y), 3(y), 4(y), [5(y), l_4]\rangle.!\,WaitRep_4\rfloor^{\{l_a,l_3\}}_{l_4})\\
&\quad \xrightarrow{\nu z.\langle z\rangle:\overline{l_3}\{l_a,l_4\}}
\end{aligned}$$

After applying a sequence of reduction relations that models the verification made on the message and applying the rule (Struct P-Par1) twice the nil processes are eliminated, and applying the labeled transition $\xrightarrow{\nu z.\langle z\rangle:\overline{l_3}\{l_a l_4\}}$, which models that the destination node l_3 accepts the message sent by the attacker node and sends back a reply message. Again, we assume that in this transition trace node l_3 sends its message before node l_4:

$$\begin{aligned}
&(\left\{{}^{(req,l_1,l_3,MAC_{13},[\,])}/_{x}\right\}^{\{l_1,l_2,l_a\}} \mid \left\{{}^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_{y}\right\}^{\{l_a,l_1,l_2,l_3,l_4\}}\\
&\quad \mid \left\{{}^{(req,l_1,l_3,MAC_{13},[l_2])}/_{x'}\right\}^{\{l_2,l_1,l_a\}} \mid \left\{{}^{(l_4,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{z}\right\}^{\{l_3,l_a,l_4\}}\\
&\quad \mid \lfloor !\,WaitRep_1\rfloor^{\{l_2,l_a\}}_{l_1} \mid \lfloor P_2'\rfloor^{\{l_1,l_a\}}_{l_2} \mid \lfloor !P_A\rfloor^{\{l_1,l_2,l_3,l_4\}}_{l_a} \mid \lfloor 0 \mid !P_3\rfloor^{\{l_a,l_4\}}_{l_3}\\
&\quad \mid \lfloor \langle 1(y), 2(y), 3(y), 4(y), [5(y), l_4]\rangle.!\,WaitRep_4\rfloor^{\{l_a,l_3\}}_{l_4})\\
&\quad \equiv_{StructP-Par1},\ (\rightarrow \times 2\ \text{(EXT BroadRecv1)})
\end{aligned}$$

The frame of *netw* at this time is

$\left\{{}^{(req,l_1,l_3,MAC_{13},[\,])}/_{x}\right\}^{\{l_1,l_2,l_a\}} \mid \left\{{}^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_{y}\right\}^{\{l_a,l_1,l_2,l_3,l_4\}}$
$\mid \left\{{}^{(req,l_1,l_3,MAC_{13},[l_2])}/_{x'}\right\}^{\{l_2,l_1,l_a\}} \mid \left\{{}^{(l_4,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{z}\right\}^{\{l_3,l_a,l_4\}}$. After applying the rules (Struct P-Par1) and (EXT BroadRecv1) twice we have:

$$\begin{aligned}
&(\left\{{}^{(req,l_1,l_3,MAC_{13},[\,])}/_{x}\right\}^{\{l_1,l_2,l_a\}} \mid \left\{{}^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_{y}\right\}^{\{l_a,l_1,l_2,l_3,l_4\}}\\
&\quad \mid \left\{{}^{(req,l_1,l_3,MAC_{13},[l_2])}/_{x'}\right\}^{\{l_2,l_1,l_a\}} \mid \left\{{}^{(l_4,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{z}\right\}^{\{l_3,l_a,l_4\}}\\
&\quad \mid \lfloor !\,WaitRep_1\rfloor^{\{l_2,l_a\}}_{l_1} \mid \lfloor !\,WaitRep_2\rfloor^{\{l_1,l_a\}}_{l_2}\\
&\quad \mid \left\lfloor P_A \left\{{}^{(l_4,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{z}\right\} \mid !P_A\right\rfloor^{l_1 l_2 l_3 l_4}_{l_a}\\
&\quad \mid \lfloor !P_3\rfloor^{\{l_a,l_4\}}_{l_3} \mid \left\lfloor P_4' \left\{{}^{(l_4,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{z}\right\}\right\rfloor^{\{l_a,l_3\}}_{l_4}) \equiv
\end{aligned}$$

Here P_4' is $\langle 1(y), 2(y), 3(y), 4(y), [5(y), l_4]\rangle.!\mathit{WaitRep}_4$. At this point, after receiving the reply message $(l_4, \mathit{rep}, l_1, l_3, [l_2, n, l_4], MAC_{31})$ node l_4 drops it because l_4 still has not output the request corresponding to this reply. However, the attacker node intercepts the reply.

$$\begin{aligned}
(&\{{}^{(req,l_1,l_3,MAC_{13},[\,])}/_{x}\}^{\{l_1,l_2,l_a\}} \mid \{{}^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_{y}\}^{\{l_a,l_1,l_2,l_3,l_4\}} \\
&\mid \{{}^{(req,l_1,l_3,MAC_{13},[l_2])}/_{x'}\}^{\{l_2,l_1,l_a\}} \mid \{{}^{(l_4,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{z}\}^{\{l_3,l_a,l_4\}} \\
&\mid \lfloor !\mathit{WaitRep}_1\rfloor_{l_1}^{\{l_2,l_a\}} \mid \lfloor !\mathit{WaitRep}_2\rfloor_{l_2}^{\{l_1,l_a\}} \\
&\mid \lfloor \langle l_1, 2(z), 3(z), 4(z), 5(z), 6(z)\rangle \mid !P_A\rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \\
&\mid \lfloor !P_3\rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4'\rfloor_{l_4}^{\{l_a,l_3\}}) \xrightarrow{\nu w.\langle w\rangle:\overline{l_a}\{l_1 l_2 l_3 l_4\}}
\end{aligned}$$

Then the attacker node l_a forwards the reply message $(l_1, \mathit{rep}, l_1, l_3, [l_2, n, l_4], MAC_{31})$ to node l_1 in the name of node l_2. This message can be overheared by the neighbors of the node l_a.

$$\begin{aligned}
(&\{{}^{(req,l_1,l_3,MAC_{13},[\,])}/_{x}\}^{\{l_1,l_2,l_a\}} \mid \{{}^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_{y}\}^{\{l_a,l_1,l_2,l_3,l_4\}} \\
&\mid \{{}^{(req,l_1,l_3,MAC_{13},[l_2])}/_{x'}\}^{\{l_2,l_1,l_a\}} \mid \{{}^{(l_4,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{z}\}^{\{l_3,l_a,l_4\}} \\
&\mid \{{}^{(l_1,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{w}\}^{\{l_a,l_1,l_2,l_3,l_4\}} \\
&\mid \lfloor !\mathit{WaitRep}_1\rfloor_{l_1}^{\{l_2,l_a\}} \mid \lfloor !\mathit{WaitRep}_2\rfloor_{l_2}^{\{l_1,l_a\}} \\
&\mid \lfloor 0 \mid !P_A\rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \mid \lfloor !P_3\rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4'\rfloor_{l_4}^{\{l_a,l_3\}}) \equiv_{PAR\text{-}0}, \rightarrow^{*}
\end{aligned}$$

The sent reply message $(l_1, \mathit{rep}, l_1, l_3, [l_2, n, l_4], MAC_{31})$ is then overheared by nodes l_1, l_2, l_3, and l_4. However, nodes l_2, l_3, and l_4 drops it because they are not the addressee but node l_1.

$$\begin{aligned}
(&\{{}^{(req,l_1,l_3,MAC_{13},[\,])}/_{x}\}^{\{l_1,l_2,l_a\}} \mid \{{}^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_{y}\}^{\{l_a,l_1,l_2,l_3,l_4\}} \\
&\mid \{{}^{(req,l_1,l_3,MAC_{13},[l_2])}/_{x'}\}^{\{l_2,l_1,l_a\}} \mid \{{}^{(l_4,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{z}\}^{\{l_3,l_a,l_4\}} \\
&\mid \{{}^{(l_1,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{w}\}^{\{l_a,l_1,l_2,l_3,l_4\}} \\
&\mid \lfloor \mathit{WaitRep}_1\,\{{}^{(l_1,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{w}\} \mid !\mathit{WaitRep}_1\rfloor_{l_1}^{\{l_2,l_a\}} \\
&\mid \lfloor !\mathit{WaitRep}_2\rfloor_{l_2}^{\{l_1,l_a\}} \mid \lfloor !P_A\rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \mid \lfloor !P_3\rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4'\rfloor_{l_4}^{\{l_a,l_3\}}) \rightarrow^{*}
\end{aligned}$$

After receiving the message $(l_1, \mathit{rep}, l_1, l_3, [l_2, n, l_4], MAC_{31})$ the source node l_1 makes verification steps on it. According to the operation of the protocol all verification steps made by l_1 pass and thus the term $ACCEPT$ is being to output.

$$\begin{aligned}
(&\{{}^{(req,l_1,l_3,MAC_{13},[\,])}/_{x}\}^{\{l_1,l_2,l_a\}} \mid \{{}^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_{y}\}^{\{l_a,l_1,l_2,l_3,l_4\}} \\
&\mid \{{}^{(req,l_1,l_3,MAC_{13},[l_2])}/_{x'}\}^{\{l_2,l_1,l_a\}} \mid \{{}^{(l_4,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{z}\}^{\{l_3,l_a,l_4\}} \\
&\mid \{{}^{(l_1,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{w}\}^{\{l_a,l_1,l_2,l_3,l_4\}} \\
&\mid \lfloor \langle ACCEPT\rangle \mid !\mathit{WaitRep}_1\rfloor_{l_1}^{\{l_2,l_a\}} \mid \lfloor !\mathit{WaitRep}_2\rfloor_{l_2}^{\{l_1,l_a\}} \\
&\mid \lfloor !P_A\rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \mid \lfloor !P_3\rfloor_{l_3}^{\{l_a,l_4\}} \mid \lfloor P_4'\rfloor_{l_4}^{\{l_a,l_3\}}) \xrightarrow{\nu v.\langle v\rangle:\overline{l_1}\{l_2 l_a\}}
\end{aligned}$$

After the source node l_1 receives the rely message sent by the attacker it sees that it is the addressee. Hence, it makes verification steps. All verification steps pass so that it outputs the special term $ACCEPT$.

$$\begin{aligned}
(&\{{}^{(req,l_1,l_3,MAC_{13},[\,])}/_{x}\}^{\{l_1,l_2,l_a\}} \mid \{{}^{(req,l_1,l_3,MAC_{13},[l_2,n,l_4])}/_{y}\}^{\{l_a,l_1,l_2,l_3,l_4\}} \\
&\mid \{{}^{(req,l_1,l_3,MAC_{13},[l_2])}/_{x'}\}^{\{l_2,l_1,l_a\}} \mid \{{}^{(l_4,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{z}\}^{\{l_3,l_a,l_4\}} \\
&\mid \{{}^{(l_1,rep,l_1,l_3,[l_2,n,l_4],MAC_{31})}/_{w}\}^{\{l_a,l_1,l_2,l_3,l_4\}} \mid \{{}^{ACCEPT}/_{v}\}^{l_1 l_2 l_a} \\
&\mid \lfloor 0 \mid !\mathit{WaitRep}_1\rfloor_{l_1}^{\{l_2,l_a\}} \mid \lfloor !\mathit{WaitRep}_2\rfloor_{l_2}^{\{l_1,l_a\}} \mid \lfloor !P_A\rfloor_{l_a}^{\{l_1,l_2,l_3,l_4\}} \mid \lfloor !P_3\rfloor_{l_3}^{\{l_a,l_4\}}
\end{aligned}$$

$| \lfloor P_4' \rfloor_{l_4}^{\{l_a, l_3\}}$).

We note that in this section we intend to demonstrate the applicability of the *sr*-calculus for reasoning about secure routing protocols and for shorter presentation purpose the proof illustrates the following attack scenario: When the attacker node receives the request message (req, l_1, l_3, MAC_{13},[ ]) from node l_1, it creates some new fake node indentifier n, then adds l_2, n and the identifier of N_4, l_4 to the list [ ], and re-broadcasts (req, 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 makes by l_3. Then, node l_3 generates the reply (l_4, rep, l_1, l_3,MAC_{31}) and sends 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 *netw* node l_1 accepts the returned invalid route $[l_2, n, l_4]$ and outputs $ACCEPT$ by the $\xrightarrow{\overline{l_1}\{l_2 l_a\}}$ transition relation. However, in $netw_{spec}$ node l_1 does not accept the returned route, thus, $ACCEPT$ is not output. Formally, at this point $netw_{spec}$ cannot perform the transition $\xrightarrow{\nu v.\langle v\rangle:\overline{l_1}\{l_2 l_a\}}$, which violates the third point of Definition 11. In this proof $netw_{spec}$, which is the ideal version of *netw*, can perform the same labelled transitions and reduction relations as *netw* except the last transition $\xrightarrow{\nu v.\langle v\rangle:\overline{l_1}\{l_2 l_a\}}$.

Finally, we note that we can easily extend the proof so that it illustrates the scenario in which the attacker receives the request message (req, l_1, l_3, MAC_{13},$[l_2]$) from node l_2, it creates some new fake node indentifier n, then adds n and the identifier of N_4, l_4 to the list [ ], and re-broadcasts (req, l_1,l_3,MAC_{13},$[l_2, n, l_4]$). The rest part of the attack scenario is the same as the scenario above. □

7 Weaker definition of security: up to barb $\Downarrow ACCEPT$

One can feel that the notion of labeled bisimilarity may be too strict for defining security because it requires the existence of same outputs in the two networks. In fact, we only need that for the same topology and attacker behavior trace the real and specification networks always do the same with respect to the barb $ACCEPT$. Namely, if the specification can/cannot emit the term $ACCEPT$ then so do the real system, and vice versa. Then this must be true for all the possible topologies and attackers.

We note that this kind of definition is not necessarily weaker than the labeled bisimilarity. It depends on the definition of the specification version, and how much it differs from the real system.

8 A systematic proof technique based on backward deduction

The proof shown in Section 6.1 is based on a forward search/reasoning, namely, we specify a certain network topology and "simulate" the operation of the protocol on this topology. The main drawback of this proof technique is that we need to take into account a huge number of possible behavior scenarios as well as many possible network topologies.

We develope a more systematic proof technique, that enables us to reason about the security of routing protocols in a more efficient way. This proof technique is based on backward reasoning, namely, we 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 could this happen. In case we get a contradiction it means that the starting assumption could not be valid, and the protocol is secure. Like in the forward search technique, this proof technique is also based on Definition 11 but we perform the labelled transitions backward, starting from the extended networks that represent the final states of system and get back into the networks that specify the initial states.

For this backward reasoning method, we define an ideal and a real system a bit differently compared to the systems that we used in the forward search. First of all, the source node in both systems will output the function term $accept(t_{list})$, instead of outputting the constant term $ACCEPT$. This means that the source has accepted the returned list t_{list}.

The procedure of the backward reasoning (backward deduction) is performed according to the protocol specification. Namely, we step backward node-by-node from the source to the destination (in the reply direction), and thereafter, from the destination to the source (in the request direction) through a possible route, which can either be the route represented by t_{list} in $accept(t_{list})$ or an another route. Formally, let us denote the state when the source has accepted the list t_{list} by the extended network $E_{accept(t_{list})}$, and the state when the source has just sent the initial request by $E_{reqinit}$. The backward deduction is based on performing labelled transitions backward from $E_{accept(t_{list})}$ to $E_{reqinit}$. We use the upper index *real* ($E^{real}_{reqinit}$, $E^{real}_{accept(t_{list})}$) and *ideal* ($E^{ideal}_{reqinit}$, $E^{ideal}_{accept(t_{list})}$) to denote the corresponding network states in the real and ideal systems, respectively. The backward deduction is based on backward application of labelled transitions from $E_{accept(t_{list})}$ to $E_{reqinit}$:

$$E_{accept(t_{list})} \;{}^{*}\!\longleftarrow \xleftarrow{\alpha_1} {}^{*}\!\longleftarrow \ldots {}^{*}\!\longleftarrow \xleftarrow{\alpha_n} {}^{*}\!\longleftarrow E_{reqinit},$$

where $\alpha_1,\ldots,\ \alpha_n$ 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 source has sent the request, to the state when the destination sends back the reply:

$$\begin{array}{l} E_{dstsentREP} \xleftarrow{\nu z.\langle z\rangle:\ \overline{l_{dst}}\{\ l_{int},\ldots\}} {}^{*}\!\longleftarrow E_{dstrecvdREQ} \xleftarrow{(t_{req})^{\sigma 2}:\{l_{dst}\in\sigma 2\}} E_{intsentREQ} \\ \xleftarrow{\nu y.\langle y\rangle:\ \overline{l_{int}}\{\ l_{dst},l_{src},\ \ldots\}} {}^{*}\!\longleftarrow E_{intrecvdREQ} \xleftarrow{(t_{reqinit})^{\sigma 1}:\{l_{int}\in\sigma 1\}} E_{srcsentREQINIT} \xleftarrow{\nu x.\langle x\rangle:\ \overline{l_{src}}\{\ l_{int},\ldots\}} \\ {}^{*}\!\longleftarrow E_{reqinit}, \end{array}$$

where the frame of $E_{dstsentREP}$ contains the substitution $\{t_{rep}/z\}$, and the frames of $E_{intsentREQ}$ and $E_{intsentREQINIT}$ contain $\{t_{req}/y\}$ and $\{t_{reqinit}/x\}$, respectively. $\sigma 1$ and $\sigma 2$ are the neighborhood of l_{int} and l_{dst}, respectively. Intuitively, the trace

$$E_{dstsentREP} \xleftarrow{\nu z.\langle z\rangle:\ \overline{l_{dst}}\{\ l_{int},\ldots\}} {}^{*}\!\longleftarrow E_{dstrecvdREQ} \xleftarrow{(t_{req})^{\sigma 2}:\{l_{dst}\in\sigma 2\}} E_{intsentREQ}$$

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

$$E_{intsentREQ} \xleftarrow{\nu y.\langle y\rangle:\ \overline{l_{int}}\{\ l_{dst},l_{src},\ \ldots\}} {}^{*}\!\longleftarrow E_{intrecvdREQ} \xleftarrow{(t_{reqinit})^{\sigma 1}:\{l_{int}\in\sigma 1\}} E_{srcsentREQINIT}$$

says that in order to node l_{int} can send the request t_{req}, it should receive the request $t_{reqinit}$ from l_{src}. Finally, the trace

$$E^{src}_{sentREQINIT} \xleftarrow{\nu x.\langle x\rangle:\ \overline{l_{src}}\{\ l_{int},\ldots\}} {}^{*}\!\longleftarrow E^{src}_{reqinit}$$

says that in order to node l_{src} can send the request $t_{reqinit}$, it should be able to composed this request somehow.

The whole example trace corresponds to the following scenario: On the route $l_{src} - l_{int} - l_{dst}$, (1) l_{src} broadcasts $t_{reqinit}$; (2) l_{int} received $t_{reqinit}$, performs calculations, and broadcasts t_{req}; (3) l_{dst} received t_{req}, performs verifications, and returns t_{rep}.

8.1 BDSR: The backward deduction algorithm for source routing protocols

Because in the paper we will refer to this algorithm many times, we give it a name BDSR algorithm, which contains the first letters of the words Backward, Deduction, and Source Routing. At the beginning, a reply including an invalid route $t_{list} = [l_1,\ldots,\ l_n]$ is assumed to be accepted by the source. Afterwards, we follow the way of this reply in a backward manner. The possible paths of this reply is 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 we are aware of the information of what message should the attacker forward so that it will be accepted later, that is, what 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.

The attacker is able to compose a reply or request message using its computational ability and knowledge base. We note that while the computation ability of the attacker is fix, its knowledge base is continually updated during the route discovery. 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 initial knowledge? (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 initial knowledge? (iii) How the topology should be formed such that the attacker is able to intercept the required message parts?

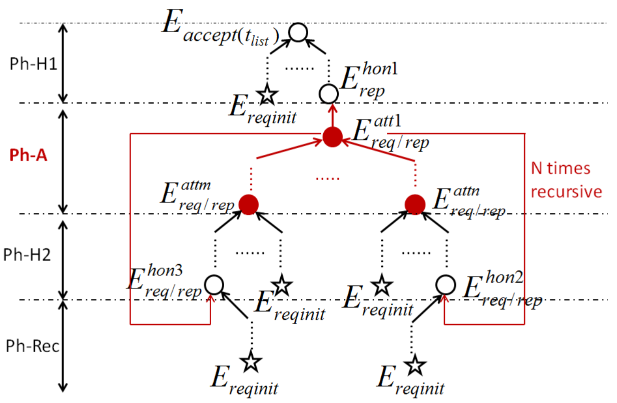


Figure 9: *The figure illustrates the main phases of the backward deduction procedure. The circles and asterisks represent the particular states during the deduction. Each state $state_i$ is represented by an extended network $E^i_{req/rep}$, which we get from $E_{accept(t_{list})}$, which represents the initial state, after performing a series of transitions, in a backward direction. The asterisks represent the terminal states, which is $E_{reqinit}$. Each honest phase contains at least one state, while attacker phases may contain zero state in them. After state $E^{hon2}_{req/rep}$, we can get into the phases Ph-A and Ph-H2 again and again, because there can be several attacker nodes or the attacker can take place in interleaving route discovery sessions.*

The backward deduction procedure can be illustrated as a derivation tree, in which the state $E_{accept(t_{list})}$ is the root. The edges are the labelled transitions which can be a broadcast and unicast send/receive, and the silent actions. The leaves of the tree are the terminal states, which is the extended network $E_{reqinit}$ (denoted by an asterisk). More specifically, the backward deduction procedure is accomplished as follows:

1. At the beginning, we assume that the source node has accepted the list t_{list}, which means that the function term $accept(t_{list})$ has been output by l_{src}. Formally, at first, the frames of both $E^{real}_{accept(t_{list})}$ and $E^{ideal}_{accept(t_{list})}$ contain only the substitutions represent the initial knowledge of the attackers, and $\{accept(t_{list}) \ / \ x_{accept}\}$:

$$\{accept(t_{list}) \ / \ x_{accept}, \ t^{att}_{initknow_1} \ / \ x_{initknow_1}, \ \dots, \ t^{att}_{initknow_k} \ / \ x_{initknow_k}\},$$

   The substitutions $\{t^{att}_{initknow_1} \ / \ x_{initknow_1}\}, \dots, \{t^{att}_{initknow_k} \ / \ x_{initknow_k}\}$ say that, by default, the initial knowledge of the attackers is available to the attackers. During the remaining deduction steps, we will reason about how the presence of this substitution in the frames could happen. Based on the protocol specification and the message format of the request and reply, we analyze which messages should have been sent, and by which nodes, so that these message exchanges eventually lead to the acceptance of t_{list}. At some point during the backward reasoning procedure, if we found that a message t_{msg} should have been sent, then the frame at that point will be extended with the substitution $\{t_{msg} \ / \ x_{msg}\}$.

2. The backward deduction *terminates with an attack* through a given route $[l_1, \ldots, l_n]$, if this list is differs from t_{list}, and by stepping backward node-by-node in this list (from l_1 to l_n and then in a reverse direction), we successfully get back to $E_{reqinit}$ (i.e., the initial request is output by the source l_{src}) in ***every required deduction branch***. More precisely, this means that the frame of $E_{reqinit}$ is extended with the substitution $\{t_{reqinit} \ / \ x_{reqinit}\}$, where $t_{reqinit}$ is the initial request sent by the source.

In order to detect an attack we focus on the case when t_{list} is not a valid route, and we reason about how could this invalid route be accepted (if this is the case). To achieve this, during the deduction procedure we try to find the traces that get back to $E_{reqinit}$ on a route differs from t_{list}. Let us assume that t_{list} in $accept(t_{list})$ represents an invalid route. In case the backward reasoning procedure terminates through the route other than t_{list}, it means that the routing protocol is insecure because the attacker can achieve that the invalid route t_{list} is accepted, while if the deduction procedure can only terminate through the route t_{list}, then we get a contradiction since t_{list} cannot be invalid, otherwise, we could not traverse back through it. Hence, in the latter case the protocol is secure.

In this backward deduction procedure, in order to perform a systematic proof based on Definition 11, we 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_{list} by using the special function $consistent(t_{list})$, and only outputs $accept(t_{list})$ if t_{list} is a correct route from the source to the destination. To attain this, we define the ideal system such that the backward deduction can only terminate without founding an attack. Namely, for the deduction paths where we can get back to the $E_{reqinit}$ through a route differs from t_{list}, we forbid to perform the last transition in one deductution branch:

$$E_{srcsentREQINIT} \overset{\nu x.\langle x\rangle :\ \overline{l_{src}}\sigma}{\longleftarrow} {}^{*}\longleftarrow E_{srcreqinit} \qquad (\textit{last-TRANS}),$$

which models the broadcast of the initial request.

To prove the security of on-demand source routing protocols based on the backward deduction approach, we apply the Definition 11 in a reverse direction, specifically:

Definition 14. *Let $E^{real}_{accept(t_{list})}$ and $E^{ideal}_{accept(t_{list})}$ be the real and the ideal specification variants of a (on-demand source) routing protocol Prot in the sr-calculus. The protocol Prot is said to be secure if for all the possible routes represented by the list t_{list}, the following holds:*

1. $E^{real}_{accept(t_{list})} \approx_s E^{ideal}_{accept(t_{list})}$;

2. *if* $E^{real}_{accept(t_{list})} \longleftarrow E^{realprv}_{accept(t_{list})}$, *then* $E^{ideal}_{accept(t_{list})} \ {}^{*}\longleftarrow E^{idealprv}_{accept(t_{list})}$ *and* $E^{realprv}_{accept(t_{list})} \ \Re \ E^{idealprv}_{accept(t_{list})}$ *for some* $E^{idealprv}_{accept(t_{list})}$;

3. *if* $E^{real}_{accept(t_{list})} \overset{\alpha}{\longleftarrow} E^{realprv}_{accept(t_{list})}$ *and* $fv(\alpha) \subseteq dom(E^{realprv}_{accept(t_{list})})$ *and* $bn(\alpha) \cap fn(E^{ideal}_{accept(t_{list})}) = \emptyset$; *then* $E^{ideal}_{accept(t_{list})} \ {}^{*}\longleftarrow \overset{\alpha}{\longleftarrow} {}^{*}\longleftarrow E^{idealprv}_{accept(t_{list})}$ *and* $E^{realprv}_{accept(t_{list})} \ \Re \ E^{idealprv}_{accept(t_{list})}$ *for some* $E^{idealprv}_{accept(t_{list})}$, *where α can be a broadcast, an unicast, or a receive action.*

Intuitively, Definition 14 says that starting from the states $E^{real}_{accept(t_{list})}$ and $E^{ideal}_{accept(t_{list})}$, if the two backward deduction procedures cannot be distinguished from each other then the routing protocol is secure. In other words, the deduction of the ideal system can simulate every deduction trace of the real system. The rationality behind this Definition is based on the fact that the backward deduction of the ideal and the real systems differ only at the last transition (*last-TRANS*), which is allowed in the ideal case only when the deduction trace (so far) conforms with the list t_{list}, while it is allowed in the real case for every possible trace. Hence, the deduction of the ideal system, $E^{ideal}_{accept(t_{list})}$, can simulate the deduction of $E^{real}_{accept(t_{list})}$ if in both cases (*last-TRANS*) can only be performed when the deduction trace confoms with t_{list}. This means that t_{list}, which is accepted at the end of the route discovery, must be valid. The ideal system is defined such that the accepted list t_{list} must always be valid, and if we deduce that this is also true for the real system, then the routing protocol is secure.

In phase *Ph-H1* we investigate how the reply could propagate. The backward deduction may not involve the attacker interference in both the request and reply directions. Hence, the attacker phase *Ph-A* can be skipped, that is, it contains zero state. The extended networks with the upper index *hon* represents the states where the request and reply messages are just sent by a honest node, while the networks with index *att* describe the state after an attacker node perfoms an internal computation or a message output/input. In the "honest" phases *Ph-H1*, *Ph-H2* the request and reply messages are forwarded by only the honest nodes. These phases include labelled transitions that models the send and receive actions by honest nodes, as well as the silent transitions that specify verification steps made on the received message (which are not visible for the attackers). Every step during the backward deduction in *Ph-H1* and *Ph-H2* is based on the specification of the honest nodes N_{src}, N^{i}_{int} and N_{dst}. From the state E^{hon1}_{rep} in *Ph-H1* we get into the state $E^{att1}_{req/rep}$ in *Ph-A*, which means that in the extended network $E^{att1}_{req/rep}$, an attacker node sent a reply/request, and in the resulted extended network E^{hon1}_{rep}, this message is received by a honest node. Similarly, if there is an edge from $E^{attn}_{req/rep}$ to $E^{hon2}_{req/rep}$, then this means that in $E^{hon2}_{req/rep}$ a honest node has sent a request or reply, and in the next step, an attacker node receives/intercepts it in $E^{attn}_{req/rep}$.

The reasoning about how attacker nodes could generate an incorrect reply *Rep* or request *Req* which leads to a successful attack, takes place in phase *Ph-A*. In *Ph-A*, we examine how the attacker could generate the Req/Rep message that leads to a successful attack. In particular, how the attacker can obtain or compute all the parts of the request or reply messages. The attacker(s) can obtain each part of a request/reply by either computes it based on the available information and the computation ability, or receives/intercepts a message that contains this part. In the latter case, the deduction procedure is continued with the phase *Ph-H2*, where we check how the attackers could receives/intercepts those messages. 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 or the request direction. 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 steps we deduce how this message could have been composed. This phase include labelled transitions that applies when the attacker receives/intercepts or sends a message, and silent transitions that models the computations that are performed by the attacker. An attack scenario is found when the deduction terminates (i.e., the state $E_{reqinit}$ is reached) through an route differs from t_{list}.

We let both t_{attrep} and t_{attreq} 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 ($rreq/rrep : T_{req/rep}$, $s : T_{str}$, $d : T_{str}$, $sID : T_{str}$), 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 is that we want to give mode possibility to 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.

1. First of all, we examines whether the attack could be performed if the attacker has forwarded the request/reply $t_{req/rep}$, by following to the protocol correctly. This means that in the Figure 9, the networks (or states) $E^{att1}_{req/rep}$ and $E^{attn}_{req/rep}$ are the same, and basically, the phase *Ph-A* contains only one state $E^{att}_{req/rep}$. If with this condition, the deduction terminates through an route differ from t_{list}, then an attack scenario is detected. In particular, when an attack scenario is detected we successfully prove the violation of Definition 14.

2. Otherwise, if the deduction in the first point can terminate only through t_{list}, we return to examine how the attacker can obtain or compute each part of $t_{req/rep}$, where $t_{req/rep}$ = $(head; v_1; \ldots; [List]; \ldots; v_k)$. An attack scenario is detected only when every elements of $t_{req/rep}$ can be computed/obtained by the attacker(s).

   We define *priority/weight* on terms of different types. We 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 in line 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 list and node IDs. Within the same priority terms are classified by *weight*, which specifies the number of variables and constants in them. The more subterms (names, variables, functions) are included the larger the weight is.

   Let W be a set that contains the terms to be examined whether they can be computed or obtained by the attackers.

   (I.) First, $t_{req/rep}$ is decomposed and the resulted parts *head*, v_1, ..., $[List]$, 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.) For the name/constant terms t_n, we check whether they belong to the attacker's knowledge base, namely, $t_n \in \mathcal{K}_{att}$. For each function term t_f we examine if can the attacker compute it using the current knowledge base $\mathcal{K}_{att}$ (e.g., keys for crypto functions). If there is a term $t_{n/f}$ in W that the attacker cannot compute/obtain based on its current knowledge base then we go on with step (IV.).

   (IV.) For each term $t_{n/f}$ cannot compute by the attacker, we analyse how the attacker can intercept/receive this term from either a honest node or an another attacker node. In particular, we replace (in a type conform way) some part of $(head; v_1; \ldots; [List]; \ldots; v_k)$ with $t_{n/f}$, then we reason about how this modified message could propagate during the route discovery. If for every $t_{n/f}$ the deduction can terminate only through t_{list}, then the protocol is secure, otherwise, an attack scenario is found.

During the deduction procedure, whenever we get into a loop, namely, we reach the term that has already been examined before, then we finish that branch of deduction to avoid infinite loop. During the backward deduction, we also keep track of the topology $\mathcal{T}_{top}$, which is the topology belongs to an attack scenario (if any). At first, $\mathcal{T}_{top}$ does not contain any edges, and first state $E_{accept(t_{list})}$ is

$$E_{accept(t_{list})} \overset{def}{=} \{accept(t_{list})/x_{accept}\}^{\sigma_{src} \cup \{l_a\}} \mid \lfloor !P_{src} \rfloor^{\sigma_{src}}_{l_{src}} \mid \textstyle\prod_{i \in 1,\ldots,n} \lfloor !P^i_{int} \rfloor^{\sigma^i_{int}}_{l^i_{int}} \mid \lfloor !P_{dst} \rfloor^{\sigma_{dst}}_{l_{dst}}.$$

where σ_{src}, σ^i_{int} and σ_{dst} are empty. During the backward deduction, if at one point we found that to make the source accepts t_{list}, the source l_{src} should send a message $t_{req/rep}$ to l_i, then we update $\mathcal{T}_{top}$ with the new edge between l_{src} and l_i. Further, we add l_i to the neighborhood of l_{src}, σ_{src}, and add l_{src} to σ_i (assuming bi-directional edges). $\{accept(t_{list})/x_{accept}\}^{\sigma_{src} \cup \{l_a\}}$ says that the output $accept(t_{list})$ is available to the neghborhood 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:

$$E_{reqinit} \overset{def}{=} \{accept(t_{list})/x_{accept}\}^{\sigma_{src}\cup\{l_a\}}, \ldots, \{t_{reqinit}/x_{reqinit}\}^{\sigma_{src}\cup\{l_a\}} \mid \lfloor !P_{src} \mid P_{src}^{reqinit} \rfloor_{l_{src}}^{\sigma_{src}} \mid \prod_{i\in 1,\ldots,n} \lfloor !P_{int}^{i} \rfloor_{l_{int}^{i}}^{\sigma_{int}^{i}} \mid \lfloor !P_{dst} \rfloor_{l_{dst}}^{\sigma_{dst}}.$$

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

8.2 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, we provide a general specificaion 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.

$$E_{routing} \overset{def}{=} \lfloor !P_{src} \rfloor_{l_{src}}^{\sigma_{src}} \mid \prod_{i\in 1,\ldots,n} \lfloor !P_{int}^{i} \rfloor_{l_{int}^{i}}^{\sigma_{int}^{i}} \mid \lfloor !P_{dst} \rfloor_{l_{dst}}^{\sigma_{dst}}.$$

where $N_{src} = \lfloor !P_{src} \rfloor_{l_{src}}^{\sigma_{src}}$, N_{int}^{i}, $N_{int}^{i} = \lfloor !P_{int}^{i} \rfloor_{l_{int}^{i}}^{\sigma_{int}^{i}}$, represents the i-th intermediate node, and $\prod_{i\in 1,\ldots,n} \lfloor !P_{int}^{i} \rfloor_{l_{int}^{i}}^{\sigma_{int}^{i}}$ represents the parallel composition of n intermediate nodes $\lfloor !P_{int}^{i} \rfloor_{l_{int}^{i}}^{\sigma_{int}^{i}}$, for $i \in \{1,\ldots,n\}$, N_{dst}=$\lfloor !P_{dst} \rfloor_{l_{dst}}^{\sigma_{dst}}$. Processes P_{src}, P_{int}, P_{dst} model the operation of honest nodes. We do not need to include explicitly the behavior of the attacker node(s). The attackers is modelled by the wireless environment in an implicit way, which can be seen as a cooperation of several attackers. In case an attack 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.

We note that the specific structure of each process depends on the specific routing protocol.

8.3 Analysing the security of Ariadne

In this subsection, we analyse the security of the Ariadne protocol using the backward deduction technique. We 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. We 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, we use the binary function symbols **sign** and **checksign**, with the following equations:

$$sign\,(t_{msg}, sk(l)).$$
$$checksign\,(sign\,(t_{msg}, sk(l))\,, pk(l)) = t_{msg}.$$

note that in *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 $h(t_{msg}^{1}) = h(t_{msg}^{2})$ holds only when $t_{msg}^{1} = t_{msg}^{2}$ ensures that h is collision-free. With these functions, the Ariadne protocol can be specified in the *sr*-calculus as follows:

$$E_{adriadne} \overset{def}{=} \lfloor !P_{src} \rfloor^{\sigma_{src}}_{l_{src}} \mid \prod_{i \in 1,\ldots,n} \lfloor !P^{i}_{int} \rfloor^{\sigma^{i}_{int}}_{l^{i}_{int}} \mid \lfloor !P_{dst} \rfloor^{\sigma_{dst}}_{l_{dst}}.$$

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.

$$\begin{aligned} P_{src} \overset{def}{=}\ & let\ head^{req} = (rreq, l_{src}, l_{dst}, ID)\ in \\ & let\ MAC_{sd} = mac\,(head^{req}, k(l_{src}, l_{dst}))\ \ in \\ & \langle (head^{req}, MAC_{sd}, [\ ], [\ ]) \rangle .!\, WaitRep_{src}. \end{aligned}$$

When the source node receives a reply, it examines whether the addresse is l_{src}, this is defined by the first construct $= l_{src}$. This construct is a shorthand of the process $(x_{addressee}).[x_{addressee} = l_{src}]$, 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 $\bigwedge_{j \in \{1,\ldots,last\}}(\ldots)$ 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 $accept(x_{idList})$ is output to signal the acceptance of the ID list x_{idList}. The function $j(x_{idList})$ returns the j-th element of the list x_{idList}.

$$\begin{aligned} & WaitRep_{src} \overset{def}{=} \\ & \quad (= l_{src}, = rrep, = l_{src}, = l_{dst}, x_{idList}, x_{sigList}, x_{sigDst}). \\ & \quad [1(x_{idList}) \in \sigma_{src}] \\ & \quad let\ head^{rep} = (rrep, l_{src}, l_{dst})\ in \\ & \quad let\ hash_0 = mac(head^{req}, k(l_{src}, l_{dst}))\ in \\ & \quad [(head^{rep}, x_{idList}, x_{sigList}) = checksign(x_{sigDst},\ pk(l_{dst}))] \\ & \quad \textstyle\bigwedge_{j \in \{1,\ldots,last\}}\ (\ let\ hash_j = h((j(x_{idList}), hash_{j-1}))\ in \\ & \quad \qquad\qquad\qquad let\ list_j = [1(x_{idList}), \ldots, j(x_{idList})]\ in \\ & \quad \qquad\qquad\qquad [(head^{req}, list_j, hash_j) = checksign(j(x_{sigList}), pk(l_{j(x_{idList})}))]\ ) \\ & \quad \langle accept(x_{idList}) \rangle. \end{aligned}$$

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

$$\begin{aligned} P^{i}_{int} \overset{def}{=}\ & (= rreq, x_{src}, x_{dst}, x_{id}, x_{hashchain}, x_{idList}, x_{sigList}). \\ & [last(x_{idList}) \in \sigma^{i}_{int}] \\ & let\ head^{req} = (rreq, x_{src}, x_{dst}, x_{id})\ in \\ & let\ hash_i = h((l_i,\ x_{hashchain}))\ in \\ & let\ sig_i = sign((head^{req},\ hash_i,\ [x_{idList},\ l_i],\ x_{sigList}),\ sk(l_i))\ in \\ & \langle (head^{req}, hash_i, [x_{idList}, l_i], [x_{sigList}, sig_i]) \rangle .!\, WaitRep^{i}_{int}. \end{aligned}$$

When 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 int_i. If so, the reply is forwarded to the previous node.

$$
\begin{aligned}
&\mathit{WaitRep}^{i}_{int} \overset{def}{=} \\
&\qquad (= l_i, = \mathit{rrep}, x_{src}, x_{dst}, x_{idList}, x_{sigList}, x_{sigDst}).\\
&\qquad [\mathit{prev}(x_{idList}, l_i) \in \sigma^{i}_{int}]\ [\mathit{next}(x_{idList}, l_i) \in \sigma^{i}_{int}]\\
&\qquad \langle(\mathit{prev}(x_{idList}, l_i), \mathit{rrep}, x_{src}, x_{dst}, x_{idList}, x_{sigList}, x_{sigDst})\rangle.
\end{aligned}
$$

We note that the function $\mathit{prev}(t_{list}{:}T_{list},\ l_i{:}T_{id})$ and $\mathit{next}(t_{list}{:}T_{list},\ l_i{:}T_{id})$ return *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 t_{list}, $\mathit{next}(t_{list}{:}T_{list},\ l_i{:}T_{id})$ returns l_{dst}, while if l_i is the first element of t_{list}, then $\mathit{prev}(t_{list}{:}T_{list},\ l_i{:}T_{id})$ returns l_{src}.

P_{dst} describes the behavior of the destination node. Within a session, when l_{dst} receives the first message it examines the type of a the message, and the ID of the destination. After that, l_{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_{dst} computes the signature on the whole packet and sends back the reply to the last node in the list, $\mathit{last}(x_{idList})$.

$$
\begin{aligned}
P_{dst} \overset{def}{=}\ & (= \mathit{rreq}, x_{src}, = l_{dst}, x_{id}, x_{hashchain}, x_{idList}, x_{sigList}).\\
& [\mathit{last}(x_{idList}) \in \sigma_{dst}]\\
& \mathit{let}\ \mathit{head}^{req} = (\mathit{rreq}, x_{src}, l_{dst}, x_{id})\ \mathit{in}\\
& \mathit{let}\ \mathit{hash}_0 = \mathit{mac}(\mathit{head}^{req}, k(x_{src}, l_{dst}))\ \mathit{in}\\
& \textstyle\bigwedge_{j\in\{1,\dots,last\}} (\ \mathit{let}\ \mathit{hash}_j = h((j(x_{idList}), \mathit{hash}_{j-1}))\ \mathit{in}\\
& \qquad\qquad\quad \mathit{let}\ \mathit{list}_j = [1(x_{idList}), \dots, j(x_{idList})]\ \mathit{in}\\
& \qquad\qquad\quad [(\mathit{head}^{rep}, \mathit{list}_j, \mathit{hash}_j) = \mathit{checksign}(j(x_{sigList}),\ pk(l_{j(x_{sigList})}))]\ )\\
& \mathit{let}\ \mathit{head}^{rep} = (\mathit{rrep}, x_{src}, l_{dst})\ \mathit{in}\\
& \mathit{let}\ \mathit{sig}_{dst} = \mathit{sign}((\mathit{head}^{rep},\ x_{hashchain},\ x_{idList},\ x_{sigList})\ \mathit{in}\\
& \langle(\mathit{last}(x_{idList}), \mathit{head}^{rep}, x_{idList}, x_{sigList}, \mathit{sig}_{dst})\rangle.
\end{aligned}
$$

We will show that Definition 14 is violated with t_{list}, $t_{list} = [l_{int}, l_{att}]$, where l_{int} and l_{att} are the IDs of a honest intermediate node and an attacker node, respectively. At the beginning, the attack topology $\mathcal{T}_{top}$ is empty, while the set of the edges in the route t_{list}, $\mathcal{T}_{tlist}$, is $\{l_{src} - l_{int},\ l_{int} - l_{att},\ l_{int} - l_{dst}\}$. Specifically, the reply received by the source has the form:

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

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

1. Assuming one attacker node, in the first case, when l_{src} receives the reply *rep* from l_{int}, we have the following message exchanges: The first state (State-1 in Figure 10) describes $E_{accept(t_{list})}$, in which $\mathit{accept}(t_{list})$ has been broadcast. The source broadcasts *accept* only when l_{int} is its neighbor. This backward deduction step is described by the labelled transitions trace

   $E_{accept(t_{list})} \xleftarrow{\nu x_{acc}.\langle x_{acc}\rangle:\ \overline{l_{src}}\{\ \emptyset\}} E_{srcsentaccept}\ {}^{*}\xleftarrow{\tau} E_{srcnbrchecked} \xleftarrow{\tau} E_{srctochecknbr}$, where

   $$E_{accept(t_{list})} \overset{def}{=} \{\mathit{accept}(t_{list})\ /\ x_{acc}\}\ |\ \lfloor !P_{src}\rfloor^{\emptyset}_{l_{src}}\ |\ \textstyle\prod_{i\in 1,\dots,n} \lfloor !P^{i}_{int}\rfloor^{\emptyset}_{l^{i}_{int}}\ |\ \lfloor !P_{dst}\rfloor^{\emptyset}_{l_{dst}}.$$

   $\xleftarrow{\nu x_{acc}.\langle x_{acc}\rangle:\ \overline{l_{src}}\{\ \emptyset\}}$ says that at the beginning l_{src} does not have any neighbor. The state $E_{srcsentaccept}$ is the same as $E_{accept(t_{list})}$, except that the source node is $\lfloor\langle \mathit{accept}(t_{list})\rangle|!P_{src}\rfloor^{\emptyset}_{l_{src}}$. The series of silent transitions represent the verification of per-hop signatures performed by l_{src}, and this trace ends with the neighbor check. In $E_{srcnbrchecked}$ the source has just performed a neighbor check, that is, the source process is $\lfloor \mathit{SigVerifs}\ \langle \mathit{accept}(t_{list})\rangle|!P_{src}\rfloor^{l_{int}}_{l_{src}}$,

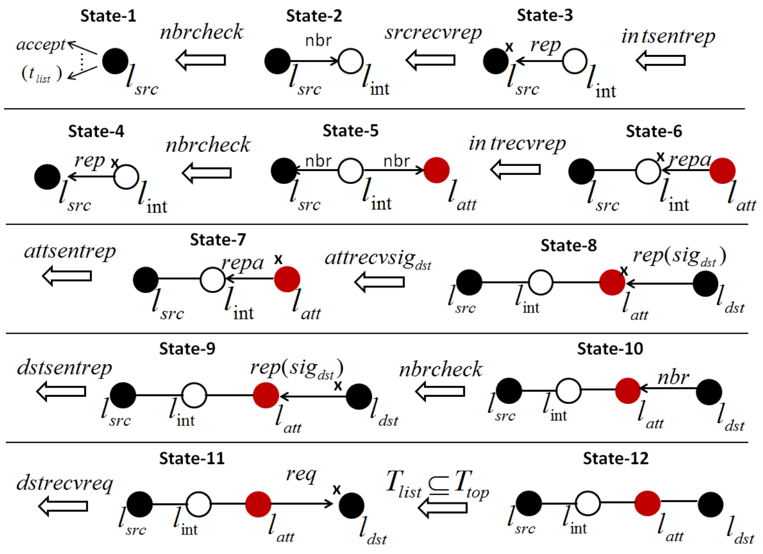


Figure 10: *Analysing Ariadne: The backward deduction steps based on the first case. Note that in the figure, the signature and hash verification steps are not illlustrated, which happen after nbrcheck and before a message is sent. In the sr-calculus, signature verifications is modelled by a series of silent transitions.*

where *SigVerifs* is the code part right after the neighbor check. At this point, the neighborhood of l_{src} is updated with l_{int}, because the term accept can only output at the end when the neighbor check succeeds. Finally, in $E_{srctochecknbr}$ the source is about to check if the first ID belongs to its neighbor, namely: $\lfloor [l_{int} \in \sigma_{src}] SigVerifs \langle accept(t_{list}) \rangle | !P_{src} \rfloor_{l_{src}}^{l_{int}}$. The last (rightmost) silent transition in the trace corresponds to State-2 in Figure 10.

The next backward deduction steps, illustrated by State-3 and State-4 in Figure 10, represent the states in which the source receives and the intermediate node sends the reply *rep*, respectively:

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

Based on the transition system of *sr*-calculus, this is specified by the trace

$$E_{srcrevcdrep} \xleftarrow{(rep)^{\sigma_{int}}:\{l_{src} \in \sigma_{int}\}} E_{intsentrep} \xleftarrow{\nu x_{rep}.\langle x_{rep}\rangle:\ \overline{l_{int}}\{\ l_{src}\}} E_{inttosendrep}.$$

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

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

Because l_{int} performed the neighbor check based on $[l_{int}, l_{att}]$, there must be a bi-directional link between l_{int} and l_{att}. The first six states belong to the honest phase *Ph-H1* in Figure 9. In *State-7* we reason about the attacker's behavior, namely, we get into phase *Ph-A*.

In the rest part of this paper, we will explain the backward deduction informally to make it more easier to read. The formal interpretation of the deduction, based on labelled transition traces can be given in the same way as the transitions we 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 *repA*, it follows that a reply (denoted by *repdst*) must have been sent directly by the destination l_{att}, where:

$$l_{dst} \to l_{att} : repdst = (l_{att},\, rrep,\, l_{src},\, l_{dst},\, ID,\, [l_{int},\, l_{att}],\, [sig_{int},\, sig_{att}],\, sig_{dst}).$$

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

$$l_{att} \to l_{dst} : reqA = (rreq,\, l_{src},\, l_{dst},\, ID,\, hash_{att},\, [l_{int},\, l_{att}],\, [sig_{int},\, sig_{att}]).$$

However, the last two messages means that there is a bi-directional link between l_{att} and l_{dst}. At this point we have $\mathcal{T}_{tlist} \subseteq \mathcal{T}_{top}$, hence, from this point, the ideal sytem can always simulate the deduction of the real system. Consequently, this deduction branch cannot lead to an attack because the route defined by t_{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 *repA*. Recall that to be successful the attacker must obtain all the parts of *repA*. 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 *repdst*, sent by l_{dst} to l_{att} (*States-8-9*). This bring us back to the situation similar to the previous case, namely, $\mathcal{T}_{tlist} \subseteq \mathcal{T}_{top}$ becomes valid after *State-11*. To summarize, the first case cannot result in an attack scenario.

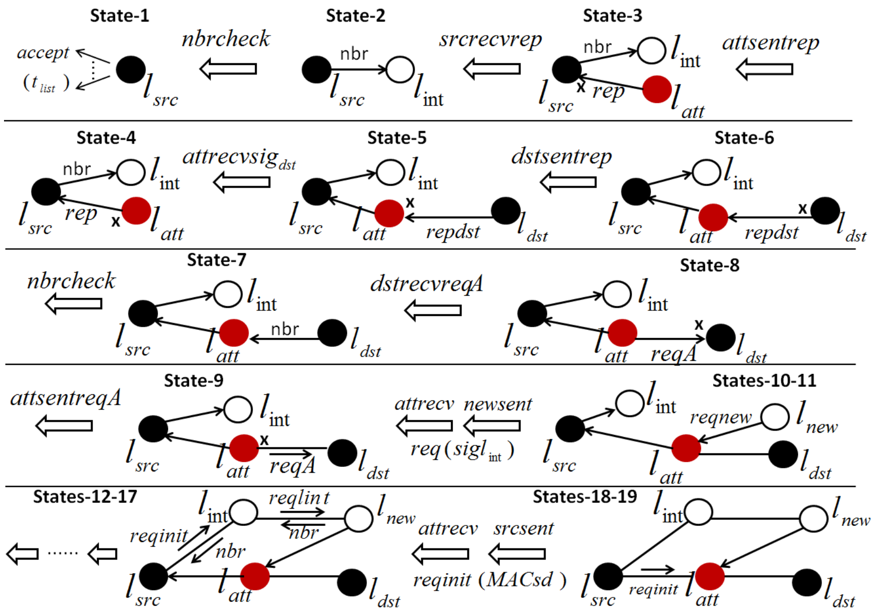


Figure 11: *Analysing Ariadne based on BDSR: The backward deduction steps based on the second case.*

2. In the second case (Figure 11), the source accepts the reply *repA* 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* $\mathcal{T}_{top}$ 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*. We 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. We examine how the attacker could compose each part of *repA* (steps I-IV of BDSR), and start with the highest priority/weight element, sig_{dst}. Like in the first case, the reply (denoted by *repdst*) must have been sent by the destination, l_{dst} to l_{att} (shown in *State-5*):

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

l_{dst} can only send *repdst* when it has already received a request, *reqA*, from l_{att} (*States-8-9*). In State-9, $\mathcal{T}_{top}$ is updated with the bi-directional link between l_{att} and l_{dst}:

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

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

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

from l_{int}, and before this, l_{int} must have received the initial request from l_{src}. This results in $\mathcal{T}_{tlist} \subseteq \mathcal{T}_{att}$. 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 IDs, 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, *reqnew*, which contains the list $[l_{int},\ l_{new}]$:

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

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

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

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

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} (illustated in *State-18-19*). The link $l_{src} - l_{att}$ is added to $\mathcal{T}_{top}$.

With these deduction branches it follows that the attacker l_{att} can obtain or compute all the parts of *repA*. Formally, on every branch, we could get back to the state $E_{reqinit}$, and $\mathcal{T}_{tlist} \nsubseteq \mathcal{T}_{top}$ is valid throughout the deduction procedure. Based on this deduction path, the Definition 14 is violated because the labelled transition traces that correspond to this backward deduction, is allowed in the real system, however, this trace cannot be simulated in the ideal system.

8.4 Analysing the security of endairA

In this subsection, we analyse the security of the endairA protocol. The endairA protocol is proposed by the authors in [3], 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 found in it. The security mechanism of endairA use less crypto functions, and the request and reply messages are protected reversely compared to the solution in Ariadne. This can found in the naming of the protocol, because endairA is the reverse of Ariadne. We start with the specification of the processes in the *sr*-calculus:

$$E_{endairA} \overset{def}{=} \lfloor !P_{src} \rfloor^{\sigma_{src}}_{l_{src}} \mid \textstyle\prod_{i \in 1,\dots,n} \lfloor !P^{i}_{int} \rfloor^{\sigma^{i}_{int}}_{l^{i}_{int}} \mid \lfloor !P_{dst} \rfloor^{\sigma_{dst}}_{l_{dst}}.$$

$$P_{src} \overset{def}{=} \nu ID.\langle (rreq, l_{src}, l_{dst}, ID, [\,], [\,]) \rangle.!\,WaitRep_{src}.$$

$$\begin{array}{l}
WaitRep_{src} \overset{def}{=} \\
\quad (= l_{src}, = rrep, = l_{src}, = l_{dst}, x_{idList}, x_{sigDst}, x_{sigList}). \\
\quad [1(x_{idList}) \in \sigma_{src}] \\
\quad let\ head^{rep} = (rrep, l_{src}, l_{dst})\ in \\
\quad [(head^{rep}, x_{idList}) = checksign(x_{sigDst},\ pk(l_{dst}))] \\
\quad let\ sigListTill_0 = [x_{sigDst}]\ in \\
\quad let\ tillsig_0 = (head^{rep}, x_{idList}, sigListTill_0)\ in \\
\quad \bigcap_{j \in \{1,\dots,last\}} \ (\ [(tillsig_{j-1} = checksign(j(x_{sigList}), pk(j(x_{idList}))] \\
\qquad\qquad\qquad let\ sig_j = sign((tillsig_{j-1}),\ sk(j(x_{idList})))\ in \\
\qquad\qquad\qquad let\ sigListTill_j = [sigListTill_{j-1},\ sig_j]\ in \\
\qquad\qquad\qquad let\ tillsig_j = (head^{rep}, x_{idList}, sigListTill_j)\ in\ ) \\
\quad \langle accept(x_{idList}) \rangle.
\end{array}$$

$$\begin{array}{l}
P^{i}_{int} \overset{def}{=} (= rreq, x_{src}, x_{dst}, x_{id}, x_{idList}). \\
\qquad [last(x_{idList}) \in \sigma^{i}_{int}]\ \langle (rreq, x_{src}, x_{dst}, x_{id}, [x_{idList}, l_i]) \rangle.!\,WaitRep^{i}_{int}.
\end{array}$$

$$\begin{array}{l}
WaitRep^{i}_{int} \overset{def}{=} \\
\quad (= l_i, = rrep, x_{src}, x_{dst}, x_{idList}, x_{sigDst}, x_{sigList}). \\
\quad [prev\,(x_{idList}, l_i) \in \sigma^{i}_{int}]\ [next\,(x_{idList}, l_i) \in \sigma^{i}_{int}] \\
\quad let\ head^{rep} = (rrep, l_{src}, l_{dst})\ in \\
\quad [(head^{rep}, x_{idList}) = checksign(x_{sigDst},\ pk(l_{dst}))] \\
\quad let\ sigListTill_0 = [x_{sigDst}]\ in \\
\quad let\ tillsig_0 = (head^{rep}, x_{idList}, sigListTill_0)\ in \\
\quad \bigcap_{j \in \{1,\dots,last\}} \ (\ [(tillsig_{j-1} = checksign(j(x_{sigList}), pk(j(x_{idList}))] \\
\qquad\qquad\qquad let\ sig_j = sign((tillsig_{j-1}),\ sk(j(x_{idList})))\ in \\
\qquad\qquad\qquad let\ sigListTill_j = [sigListTill_{j-1},\ sig_j]\ in \\
\qquad\qquad\qquad let\ tillsig_j = (head^{rep}, x_{idList}, sigListTill_j)\ in\ ) \\
\quad \langle (prev\,(x_{idList}, l_i)\,, rrep, x_{src}, x_{dst}, x_{idList}, x_{sigDst}, x_{sigList}) \rangle.
\end{array}$$

$$\begin{aligned}P_{dst} \overset{def}{=} & (= rreq, x_{src}, = l_{dst}, x_{id}, x_{idList}).\\ & [last(x_{idList}) \in \sigma_{dst}]\\ & let\ sig_{dst} = sign((rrep,\ x_{src},\ l_{dst},\ x_{idList}),\ sk(l_{dst}))\ in\\ & \langle(last(x_{idList}), rrep, x_{src}, l_{dst}, x_{idList}, sig_{dst})\rangle.\end{aligned}$$

Again, we make the analysis easy to read by discussing the the deduction steps informally. The corresponding labelled transitions, and extended networks can be given in the same way as in the analysis of the Ariadne protocol. We 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 t_{list} includes l_{att} as the first element, namely, $t_{list} = [l_{att},\ l^1_{int} \ldots,\ l^k_{int}]$ 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 t_{list} would belong to a honest node).

   $$l_{att} \to l_{src} :\ repA = (l_{src},\ rrep,\ l_{src},\ l_{dst},\ ID,\ [l_{att},\ l^1_{int} \ldots,\ l^k_{int}],\ sig_{dst},\ [sig^k_{int},\ \ldots,\ sig^1_{int},\ sig_{att}]).$$

   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 $\mathcal{T}_{tlist} \subseteq \mathcal{T}_{top}$ will hold. Namely, this will be resulted after the following backward deduction steps

   $$l_{att} \overset{rep^1}{\longleftarrow} l^1_{int} \overset{rep^2}{\longleftarrow} \ldots \overset{rep^k}{\longleftarrow} l^k_{int} \overset{rep^{dst}}{\longleftarrow} l_{dst},$$

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

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

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

   C1: sig^1_{int} can be included in *repA*. However, as already discussed in the previous point, *repA* must traverse on the path t_{list}, which results in $\mathcal{T}_{tlist} \subseteq \mathcal{T}_{top}$.

   C2: sig^1_{int} can be included in a request or reply that contains sig^1_{int} 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

   $$req' = (rreq,\ l_{src},\ l_{dst},\ sig^1_{int},\ t'_{list}).$$

   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

$$l_{att} \overset{req^{(1)}}{\longleftarrow} l^{j}_{int} \overset{req^{(2)}}{\longleftarrow} \ldots \overset{req^{(m)}}{\longleftarrow} l^{t}_{int} \overset{req^{(m+1)}}{\longleftarrow} l'_{att}$$

However, since the initial request sent by l_{src} does not allow a signature to be in the place of the session ID, l^{t}_{int} 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^{1}_{int} can be obtained).

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

To summarize our analysis: We proved that the endairA protocol is secure if we assume one attacker node.

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'_{att} and l_{att} can be two different attacker nodes. We analyse how (i.e., on which path) the second attacker node l'_{att} can obtain sig^{1}_{int}. The second attacker node l'_{att} must receive the reply sent by the honest node l^{1}_{int}, because only this reply contains sig^{1}_{int}. There can be two possibilities:

   - We have to extend the route accepted by the source (t_{list}) with l'_{att} by inserting it before l^{1}_{int}, namely, $t_{list} = [l_{att}, l'_{att}, l^{1}_{int}, \ldots, l^{k}_{int}]$. Node l'_{att} receives the following reply, denoted by *rep1*, from l^{1}_{int}:

   $$l^{1}_{int} \rightarrow l'_{att} : rep1 = (l'_{att}, rrep, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^{1}_{int}, \ldots, l^{k}_{int}], sig_{dst}, [sig^{k}_{int}, \ldots, sig^{1}_{int}]).$$

   Then, we keep track backward the possible path on which this reply has traversed, and found that from l^{1}_{int} backward to l_{dst} the reply must be forwarded correctly according to the protocol:

   $$l^{1}_{int} \overset{rep1}{\longleftarrow} \ldots \overset{repk-1}{\longleftarrow} l^{k}_{int} \overset{repk}{\longleftarrow} l_{dst},$$

   and the corresponding requests must be forwarded correctly from l^{1}_{int} to l_{dst}

   $$l^{1}_{int} \overset{req1}{\longrightarrow} \ldots \overset{reqk-1}{\longrightarrow} l^{k}_{int} \overset{reqk}{\longrightarrow} l_{dst},$$

   where $reqk = (rreq, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^{1}_{int}, \ldots, l^{k}_{int}]), \ldots, req1 = (rreq, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^{1}_{int}])$. Further, l'_{att} must have sent the request, *reqA'*, *reqA'* $= (rreq, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}])$. At this point we get into the attacker phase *Ph-A*, and reason about how node l'_{att} could compose *reqA'*. Since l'_{att} cannot generate the fresh session ID, it must be received as a part of the request sent by l_{src}. Now, the main question is that (i.) does the second attacker, l'_{att}, receive the request directly from the first attacker, l_{att}, or (ii.) from an intermediate honest node l^{0}_{int}. The first case means that there is a link between l_{att} and l'_{att}, and because the reply was sent by l_{att} to l_{src}, there is also a link between l_{att} and l_{src}. To summarize, this means that the route, $[l_{att}, l'_{att}, l^{1}_{int}, \ldots, l^{k}_{int}]$, accepted by the source is valid, thus, no attack is detected. In the second case, we have the following messages:

   $$l^{0}_{int} \longrightarrow l'_{att}\text{: } (rreq, l_{src}, l_{dst}, ID, [t'_{list}, l^{0}_{int}]),$$

   for some list t'_{list} that begins with l_{att}. We examine the case when $t'_{list} = [l_{att}]$, that is, l^{0}_{int} must have received the request $(rreq, l_{src}, l_{dst}, ID, [l_{att}])$, which must have been sent by l_{att}. In order to send this request, l_{att} must receive the initial request, $reqinit = (rreq, l_{src}, l_{dst}, ID, [\,])$ from l_{src}.

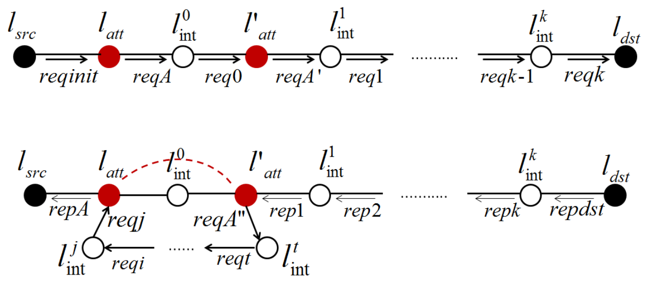


Figure 12: The figure illustrates the attack against the endairA protocol. The upper figure presents the request phase, while the one below it shows the reply phase of the attack scenario. Node IDs l_{att} and l'_{att} represent the two attacker nodes, l_{src} and l_{dst} are the IDs of the source and destination nodes, while the remaining IDs belong to the intermediate honest nodes.

To summarize, we 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_{att} and l_{att} can achieve that the source l_{src} accept the invalid route [l_{att}, l'_{att}, l^1_{int}, …, l^k_{int}] instead of the valid [l_{att}, l^0_{int}, l'_{att}, l^1_{int}, …, l^k_{int}] (shown in Figure 12).

1. *reqinit* = (*rreq*, l_{src}, l_{dst}, ID, [ ]);
2. *reqA* = (*rreq*, l_{src}, l_{dst}, ID, [l_{att}]);
3. *req0* = (*rreq*, l_{src}, l_{dst}, ID, [l_{att}, l^0_{int}]);
4. *reqA'* = (*rreq*, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}]);
5. *req1* = (*rreq*, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^1_{int}]);

   …………

   reqk = (*rreq*, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^1_{int}, …, l^k_{int}]);

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}, …, 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}, …, l^k_{int}], sig_{dst});
7. *repk* = (l^{k-1}_{int}, *rrep*, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^1_{int}, …, l^k_{int}], sig_{dst}, [sig^k_{int}]);

   …………

   rep2 = (l^1_{int}, *rrep*, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^1_{int}, …, l^k_{int}], sig_{dst}, [sig^k_{int}, …, sig^2_{int}]).
8. *rep1* = (l'_{att}, *rrep*, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^1_{int}, …, l^k_{int}], sig_{dst}, [sig^k_{int}, …, sig^1_{int}]).
9. *repA* = (l_{src}, *rrep*, l_{src}, l_{dst}, ID, [l_{att}, l'_{att}, l^1_{int}, …, l^k_{int}], sig_{dst}, [sig^k_{int}, …, sig^1_{int}, *sig'*$_{att}$, sig_{att}]).

In the reply phase, the reply is sent back by l_{dst}, l^k_{int}, …, l^1_{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} need to find an another way to forward information to l_{att}.

l'_{att} sends l_{att} the signatures sig^1_{int} in an interleaving request session with the session ID *ID'*. 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 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 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.

Interleaving$_1$: *reqA"* = (*rreq*, l_{src}, l_{dst}, sig^1_{int}, $[l'_{att}]$);

Interleaving$_2$: *reqt* = (*rreq*, l_{src}, l_{dst}, sig^1_{int}, $[l'_{att}, l^t_{int}]$);

....

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

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 sig^1_{int}, in different interleaving sessions.

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

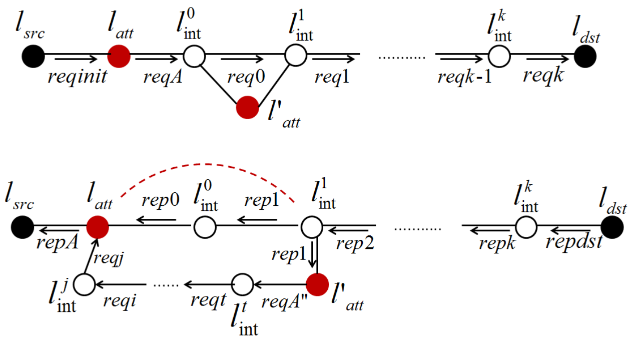


Figure 13: The figure illustrates an another possible attack against the endairA protocol. In this scenario, l'_{att} is not the part of the invalid route (t_{list}) that has been accepted by l_{src}, but it is neighbor of l^0_{int} and l^1_{int}. In particular, the two attacker nodes can achieve that l_{src} accepts the route $[l_{att}, l^1_{int}, \ldots, l^k_{int}]$ instead of the correct route $[l_{att}, l^0_{int}, l^1_{int}, \ldots, l^k_{int}]$.

In this attack, the request messages are sent by l_{att} and the honest nodes in a correct way. The attacker l'_{att} stays idle in the request phase. In addition, the corresponding reply is sent back correctly by l_{dst}, l^k_{int}, ..., l^0_{int}. Instead of sending the correct reply to l_{src}, the attacker l_{att} waits for the signatures sig_{dst}, sig^k_{int}, ..., sig^1_{int}, which can be sent by l'_{att} in interleaving sessions. In each session, l'_{att} broadcasts a request message that contains the signature(s) in place of the session ID.

9 *sr-verif*: On automating the verification

In this section, I present a novel automated verification technique for on-demand source routing protocols, based on deductive model checking and backward reasoning procedure.

9.1 Assumptions on routing protocols and attacker model

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 properties of source routing protocols are given:

Every honest node checks ID duplication in ID lists. When an intermediate node receives a request or 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 ID list it receives. Furthermore, I assume that before passing on request messages each honest intermediate node appends its own ID to the ID list, and the unicast messages include the information, such as ID, of the addressee.

Every intermediate node considers only the request it receives for the first time, further requests with the same header are dropped. Note that the destination can accept more requests, and the source can accept more replies. For increasing the efficiency of the verification, I assume that the attackers cannot obtain the secret keys of the honest nodes. Note that this assumption does not affect the results of this paper because in most cases the attacks can be performed without knowing the secret keys.

I assume *several attacker nodes* which are compromised nodes, meaning that they can perform computations like honest nodes, and posses information that honest nodes can have according the protocol. But not like 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 at the same time.

We do not assume direct communication channels between the attacker nodes. The reason is that the attackers with common links (i.e., neighbor 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.

Corollary 1. *In order to perform an attack, the attackers cannot remain idle after intercepting a reply in the reply direction.*

Proof. 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. □

9.2 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-verif* method, the method proposed in [**?**], and the method used by the Proverif tool [6].

In the ProVerif verification tool [6] the input of the tool is the specification of security protocols in the syntax of the applied π-calculus (Figure 14). The main advantage of using a calculus as a specification language is that the operation of protocols can be unequivocally and precisely modelled in it. The tool then translates the protocol specification to logic rules for performing automatic verification.

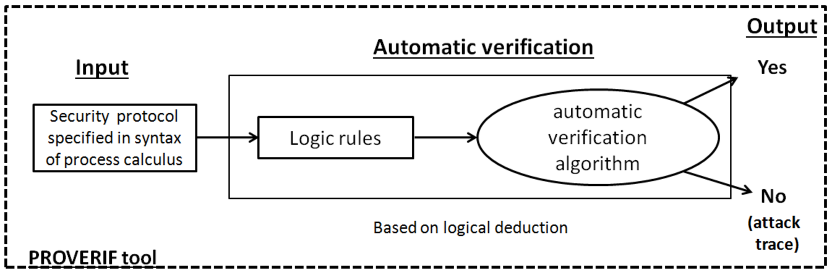

Input
Security protocol specified in syntax of process calculus
Automatic verification
Logic rules
automatic verification algorithm
Based on logical deduction
Output
Yes
No (attack trace)
PROVERIF tool

Figure 14: The concept of the ProVerif tool.

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 modelling 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 modelling 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.

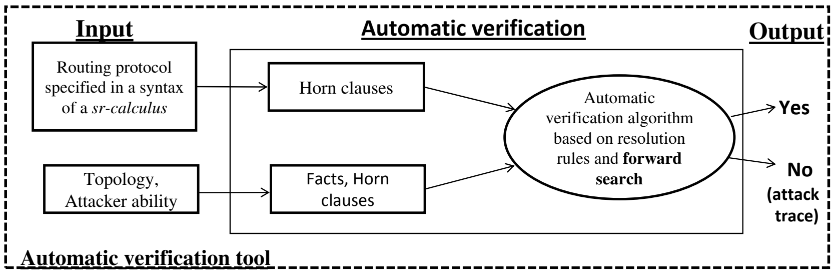

Input
Automatic verification
Output
Routing protocol specified in a syntax of a sr-calculus
Horn clauses
Automatic verification algorithm based on resolution rules and forward search
Yes
No (attack trace)
Topology, Attacker ability
Facts, Horn clauses
Automatic verification tool

Figure 15: The concept of the automatic verification method proposed in [?].

Following this concept, I proposed the verification method in [?]. As Figure 15 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. While this approach is well-suited for routing protocols and gives us a possibility of systematic proof, it has many drawbacks. One drawback of this method is that a certain topology is required to be input. Hence, to verify routing protocols one has to verify all of the possible topologies. For example, in case of n nodes there are $2^{\frac{n(n-1)}{2}}$ or $2^{n(n-1)}$ topologies, depending on if bidirectional or unidirectional edges are assumed.

It can be observed that some topologies are equivalent with each other (from the attacker's point of view) and instead of verifying all topologies only non equivalent topologies should be considered. Although taking into account the equivalent topologies reduces the complexity of verification in large scale, I achieved an even better result by proposing a *backward searching* approach that assumes an arbitrary topology at the beginning of the verification, along with a stronger attacker model. The concept of the improved method is shown in Figure 16.

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. 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 $[List_{invalid}]$. In each round, $c + 2$ ID lists are examined, where c ranges from 1 to n. In particular, an invalid ID list of c honest node IDs, $[l_1^p, \ldots, l_c^p]$, along with $c + 1$ additional lists created by inserting the attacker's unique ID into every possible places in $[l_1^p, \ldots, l_c^p]$ are analyzed. Whenever an attack is

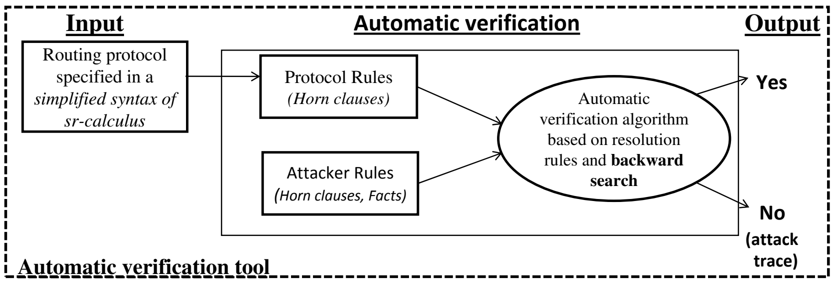

Input
Automatic verification
Output
Routing protocol specified in a *simplified syntax of sr-calculus*
Protocol Rules (*Horn clauses*)
Attacker Rules (*Horn clauses, Facts*)
Automatic verification algorithm based on resolution rules and **backward search**
Yes
No (attack trace)
Automatic verification tool

Figure 16: The concept of the automatic verification method proposed. The two main novelties compared to Figure 15 is that the verification in this case considers an arbitrary topology, and the algorithm is based on backward reasoning.

detected the tool returns the attack scenario and stops. During the searching procedure the tool follows a specific heuristic and aims at finding an attack as short as possible.

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 $l_1^p, \ldots, l_c^p$ 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 $l_1^p, \ldots, l_c^p$ in the list is disregarded. In addition, I emphasize that only the number of honest nodes in $[List_{invalid}]$ is upper bound, in order to force termination in the worst case, but the number of total nodes in the network can be arbitrary large. On the other hand, the price we have to pay is that my method, in the general case, can only prove that a routing protocol is flawless within n rounds. However in most practical examples, based on the returned information provided by the tool, users can reason about the cases beyond n. Finally, I have applied my method to verify well-known routing protocols and found that attacks were detected within only 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.

9.3 Specifying on-demand source routing protocols

In this section, we give an overview of the formal language with which the operation of routing protocols can be specified (the leftmost box in Figure 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 we introduce a variant of the *sr-calculus* with a simplified syntax, which can be edited in an easier way in text format.

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, in our method users are required to specify only the operation of one node instead of all of the nodes in the network, which is more comfortable. We note that using the existent tools designed for analysing cryptographic protocols such as the ProVerif tool, users have to define the operation of all nodes in the network, which is infeasible when the number of nodes is large.

Below the simplified syntax of the *sr-calculus* is given: *Terms* are used to specify request/reply messages and their elements, and are specified by the following syntax:

$$t ::= rreq \mid rrep \mid ID \mid l_{src}, l_{dest}, l_{att_j}, l_i \mid x_{type} \mid y_{this}, y_{prv}, y_{nxt}, y_{honprv}, y_{honnxt},\ y \mid f(t_1, \ldots, t_k).$$

The meaning of each term is as follows:

- *rreq* and *rrep* are unique constants that represent the *rreq* and *rrep* tags in route request and reply messages;
- *ID* is a name that is used to specify a message ID;

- l_{src}, l_{dest}, l_{att_j} are unique constants specifying the node ID of the source, the destination and the attacker node, respectively. Different indices in l_{att_j} represents different attacker nodes in the network.
- l_i, where $i \in \{1, \dots, n\}$, are unique constants used to specify the node IDs of the intermediate honest nodes in the network;
- x_{type} is a variable that can represent any term, that is, any term of type *type* can be bounded to it. We apply type inference during the pattern matching used in our verification method. Variables are distinguished by their type, represented by the index *types*, which can be empty, *mac*, *sign*, *senc*, *penc*, *hash*, *list*, etc. For example, x_{mac} and x_{sign} represent a variable with the type of a MAC value and a signature, respectively.
- y_{this}, y_{prv}, y_{nxt}, y_{honprv}, y_{honnxt} are variables that represent node IDs. y_{this} represents the ID of the *honest* node we are specifying, the index *this* refers to the current node. y_{prv} and y_{nxt} are variables which can be bound to either honest node IDs or l_{att_j}. The indices *prv* and *nxt* refer to the words previous and next, and are used to define the previous and next nodes in y_{this}'s point of view. y_{honprv} and y_{honnxt} are similar to y_{prv} and y_{nxt} with the only difference that they cannot be bounded to l_{att_j}. Eventually, y is a variable that covers all type of node ID-s.
- Finally, f is a (constructor) function with arity k and is used to construct terms and to model cryptographic primitives, route request and reply messages. For instance, digital signature is modelled by the function $sign(t_1, t_2)$, where t_1 models the message to be signed and t_2 models the secret key is used to sign. Route request and reply messages are modelled by the function *tuple* of k terms: $tuple(t_1,\dots,t_k)$, which we abbreviate as $(t_1,\dots,t_k)$. Function *list*(*List*) specifies the ID list, where *List* is a sequence of node IDs. We abbreviate *list*(*List*) as [*List*].

Processes model the internal operation of nodes and are specified by the following syntax:

P, Q, R ::=	*Processes*
$broadinit(t).P$	broadcast by source node
$broad(t).P$	broadcast by other nodes
$unidest((y,t)).P$	unicast by destination node
$uni((y,t)).P$	unicast by other nodes
$recvreq(t).P$	receive request
$recvrep(t).P$	receive reply
$P\|Q$	parallel composition
$(new\ n)P$	restriction
$!P$	replication
nil	nil
$letdst\ (x = g(t_1, \dots, t_n))\ in\ P\ else\ Q$	destructor application
$let\ (x = t)\ in\ P$	let
$letor\ (x = t_1)\ or\ (x = t_1)\ in\ P$	let or
$if\ nbr(y_i, y_j)\ then\ P$	neighbor
$accept([y, List])$	accepting the route $[y, List]$

The processes $\langle t\rangle.P$ in the *sr*-calculus is replaced by $broadinit(t).P$ and $broad(t).P$, respectively, in the simplified syntax. The receive action $(x).P$ is replaced by two processes $recvreq(t).P$ and $recvrep(t).P$. The processes $unidest((y_{nid}, t)).P$ and $uni((y_{nid}, t)).P$, respectively represent sending a message t to node with the ID y_{nid}. The index *nid* refers to a variable that has the node ID type.

- Both processes $broadinit(t).P$ and $broad(t).P$ first broadcast t, which is followed by the running of process P. The reason why we distinguish *broadinit* from *broad* is that the corresponding logic rules, which are basic elements of the automatic reasoning, are different.

- Processes $\mathit{unidest}((y_{nid}, t)).P$ and $\mathit{uni}((y_{nid}, t)).P$ first send (y_{nid}, t) then are continued with P. Node ID y_{nid} at the beginning of (y_{nid}, t) specifies the addressee of the message t. We note that every message sent in the network is overheared, however, honest nodes drop the message if they are not the addressee. *unidest* is used when a message is sent by the destination.

- Processes $\mathit{recvreq}(t).P$ and $\mathit{recvrep}(t).P$ wait for the term request and reply t, respectively. In case the received term is equal to t, P will run, otherwise, it stucks and 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 $(\mathit{new}\ n)P$ 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 $\mathit{letdst}\ (x = g(t_1, \ldots, t_n))\ \mathit{in}\ P\ \mathit{else}\ Q$ 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 $\mathit{checksign}(\mathit{sign}(x, \mathit{sk}(y)), \mathit{pk}(\mathit{sk}(y)))$, where the constructor $\mathit{pk}(\mathit{sk}(y))$ represents the public key generated from the given secret key *sk(y)*. 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 *checksign*, anf if the constructor function is encryption then the corresponding destructor function is decryption.

- Process $\mathit{let}\ (x = t)\ \mathit{in}\ P$ means that every occurrence of x binding t in process P.

- Process $\mathit{letor}\ (x = t_1)\ \mathit{or}\ (x = t_1)\ \mathit{in}\ P$ 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. This process is introduced for case separation purposes in the translation from a protocol specification to logic rules, given in Section 9.4.2.

- Process $\mathit{if}\ \mathit{nbr}(y_{nid_1}, y_{nid_2})\ \mathit{then}\ P$ says if node y_{nid_2} is a neighbor of node y_{nid_1} then P will run, otherwise, it stucks and idle. We emphase that in our method, by default we consider one directional edges, however, our method works in case of two directional edges as well.

- Process $\mathit{accept}([y, \mathit{List}])$ broadcasts the ID list $[y, \mathit{List}]$ 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.

We define the function $[\mathit{List}, y]$ appends the ID y to the end of the ID list $[\mathit{List}]$. We remove the syntax of networks and nodes in the simplified syntax of the *sr-calculus* because we 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_{spec}:

$$P_{spec} \overset{def}{=} !\mathit{INIT} \mid !\mathit{INTERM} \mid !\mathit{DEST}$$

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. We note that the specific structure of each process depends on the specific routing protocol, nevertheless, in general form they can be modelled in the following way:

$$\mathit{INIT} \overset{def}{=} \mathit{let}\ (y_{this} = l_{src})\ \mathit{in}\ \big((\mathit{new}\ \mathit{ID})\mathit{Prot}_{init}.\mathit{broadinit}(t_{[\ ]\in req}).!\mathit{Rep}_{init}\big).$$
$$\mathit{Rep}_{init} \overset{def}{=} \mathit{letor}\ (y_{nxt} = y_{honnxt})\ \mathit{or}\ (y_{nxt} = l_{att_i})\ \mathit{in}\ \mathit{PRep}_{init}.$$
$$\mathit{PRep}_{init} \overset{def}{=} \mathit{recvrep}((y_{this}, t_{[y_{nxt}, \mathit{List}]\in rep})).\mathit{if}\ \mathit{nbr}(y_{this}, y_{nxt})\ \mathit{then}\ \mathit{Prot}^{Rep}_{init}.\mathit{accept}([y_{nxt}, \mathit{List}]).$$

$$\mathit{INTERM} \overset{def}{=} \mathit{Interm}_1 \mid \mathit{Interm}_2$$
$$\mathit{Interm}_1 \overset{def}{=} \mathit{recvreq}(t_{[\ ]\in req}).\ \mathit{if}\ \mathit{nbr}(y_{this}, l_{src})\ \mathit{then}\ \mathit{Prot}_{interm_1}.\mathit{broad}(t_{[y_{this}]\in req}).!\mathit{Rep}_{interm_1}.$$
$$\mathit{Interm}_2 \overset{def}{=} \mathit{letor}\ (y_{prv} = y_{honprv})\ \mathit{or}\ (y_{prv} = l_{att_i})\ \mathit{in}\ \mathit{PInterm}_2.\ !\mathit{Rep}_{interm_2}.$$
$$\mathit{PInterm}_2 \overset{def}{=} \mathit{recvreq}(t_{[\mathit{List}, y_{prv}]\in req}).\mathit{if}\ \mathit{nbr}(y_{this}, y_{prv})\ \mathit{then}\ \mathit{Prot}_{interm_2}.\mathit{broad}(t_{[\mathit{List}, y_{prv}, y_{this}]\in req}).$$

$$\mathit{Rep}_{interm_1} \overset{def}{=} \mathit{Rep}_{interm^1_1} \mid \mathit{Rep}_{interm^2_1}$$
$$\mathit{Rep}_{interm^1_1} \overset{def}{=} \mathit{recvrep}((y_{this}, t_{[y_{this}]\in rep})).\ \mathit{if}\ \mathit{nbr}(y_{this}, l_{src})\ \mathit{then}\ \mathit{if}\ \mathit{nbr}(y_{this}, l_{dest})\ \mathit{then}\ \mathit{Prot}^{Rep}_{interm^1_1}.\mathit{uni}\big((l_{src}, t^1_{rep})\big).$$
$$\mathit{Rep}_{interm^2_1} \overset{def}{=} \mathit{letor}\ (y_{nxt} = y_{honnxt})\ \mathit{or}\ (y_{nxt} = l_{att_i})\ \mathit{in}\ \mathit{PRep}_{interm^2_1}.$$
$$\mathit{PRep}_{interm^2_1} \overset{def}{=} \mathit{recvrep}((y_{this}, t_{[y_{this}, y_{nxt}, \mathit{List}]\in rep})).\ \mathit{if}\ \mathit{nbr}(y_{this}, l_{src})\ \mathit{then}\ \mathit{if}\ \mathit{nbr}(y_{this}, y_{nxt})\ \mathit{then}\ \mathit{Prot}^{Rep}_{interm^2_1}.\mathit{uni}\big((l_{src}, t^2_{rep})\big).$$

$$\mathit{Rep}_{interm_2} \overset{def}{=} \mathit{Rep}_{interm^1_2} \mid \mathit{Rep}_{interm^2_2}$$
$$\mathit{Rep}_{interm^1_2} \overset{def}{=} \mathit{letor}\ (y_{prv} = y_{honprv})\ \mathit{or}\ (y_{prv} = l_{att_i})\ \mathit{in}\ \mathit{letor}\ (y_{nxt} = y_{honnxt})\ \mathit{or}\ (y_{nxt} = l_{att_i})\ \mathit{in}\ \mathit{PRep}_{interm^1_2}.$$
$$\mathit{PRep}_{interm^1_2} \overset{def}{=} \mathit{recvrep}((y_{this}, t_{[\mathit{List}_1, y_{prv}, y_{this}, y_{nxt}, \mathit{List}_2]\in rep})).\ \mathit{if}\ \mathit{nbr}(y_{this}, y_{prv})\ \mathit{then}\ \mathit{if}\ \mathit{nbr}(y_{this}, y_{nxt})\ \mathit{then}\ \mathit{Prot}^{Rep}_{interm^1_2}.\ \mathit{uni}\big((y_{prv}, t^3_{rep})\big).$$
$$\mathit{Rep}_{interm^2_2} \overset{def}{=} \mathit{letor}\ (y_{prv} = y_{honprv})\ \mathit{or}\ (y_{prv} = l_{att_i})\ \mathit{in}\ \mathit{PRep}_{interm^2_2}.$$
$$\mathit{PRep}_{interm^2_2} \overset{def}{=} \mathit{recvrep}((y_{this}, t_{[\mathit{List}, y_{prv}, y_{this}]\in rep})).\ \mathit{if}\ \mathit{nbr}(y_{this}, y_{prv})\ \mathit{then}\ \mathit{if}\ \mathit{nbr}(y_{this}, l_{dest})\ \mathit{then}\ \mathit{Prot}^{Rep}_{interm^2_2}.\ \mathit{uni}\big((y_{prv}, t^4_{rep})\big).$$

$$\mathit{DEST} \overset{def}{=} \mathit{let}\ (y_{this} = l_{dest})\ \mathit{in}\ (\mathit{letor}\ (y_{prv} = y_{honprv})\ \mathit{or}\ (y_{prv} = l_{att_i})\ \mathit{in}\ \mathit{PDEST}).$$
$$\mathit{PDEST} \overset{def}{=} \mathit{recvreq}(t_{[\mathit{List}, y_{prv}]\in req})\ \mathit{if}\ \mathit{nbr}(y_{this}, y_{prv})\ \mathit{then}\ \mathit{Prot}_{dest}.\mathit{unidest}\big((y_{prv}, t^5_{rep})\big).$$

Each process *INIT*, *INTERM* and *DEST* is composed of two parts: (i) A request part, which is placed before the process of form !*Rep*, and specifies how requests are handled; and (ii) a reply part !*Rep*, which describes how a node behaves when a reply is received. The protocol dependent procedures Prot_{init}, Prot_{interm}, Prot_{dest} and their Prot^{Rep} counterparts include neighbor checking processes of form *if* $\mathit{nbr}(y_{nid_i},\ y_{nid_j})$ *then*, functions and “inverse” functions. The notations $t_{[\mathit{List}]\in req}$ and $t_{[\mathit{List}]\in rep}$ represent the request and reply messages which include the ID list [*List*], respectively.

Process *INIT* considers the case when node y_{this} is the source, which is illustrated as *Scenario1* in the Figure 17. First, a new message ID is created, then a protocol dependent processing part Prot_{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 $t_{[\ ]\in req}$ is broadcast, which is specified by process $\mathit{broadinit}(t_{[\ ]\in req})$. Afterwards, the source permanently waits for the corresponding reply, which is modelled by process $!\mathit{Rep}_{init}$. In Rep_{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 PRep_{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.

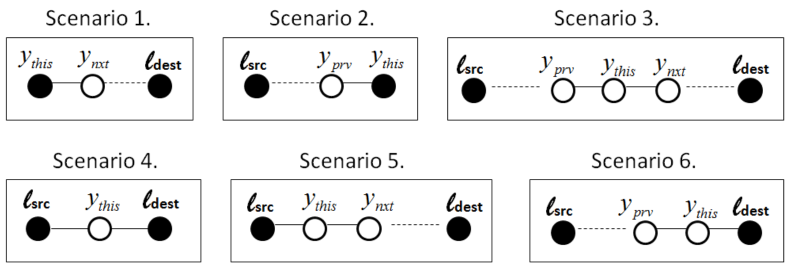

Scenario 1.
y_{this} y_{nxt} l_{dest}
Scenario 2.
l_{src} y_{prv} y_{this}
Scenario 3.
l_{src} y_{prv} y_{this} y_{nxt} l_{dest}
Scenario 4.
l_{src} y_{this} l_{dest}
Scenario 5.
l_{src} y_{this} y_{nxt} l_{dest}
Scenario 6.
l_{src} y_{prv} y_{this} l_{dest}

Figure 17: *The scenarios described by the protocol specification.*

Process *INTERM* specifies the case when node y_{this} is an intermediate node, and is divided to two subprocesses $Interm_1$ and $Interm_2$. These two processes distinguish four different scenarios *Scenario3-6*, which are shown in the Figure 17.

Process $Interm_1$ describes the case when the current node y_{this} receives the initial request. On receiving the request, y_{this} examines whether the source is its neighbor continued by re-broadcasting a request when all the protocol specific verification steps are passed, and waits for the reply. Process Rep_{interm_1} is splitted into two processes $Rep_{interm_1^1}$ and $Rep_{interm_1^2}$, where the first one specifies the *Scenario4* in the Figure 17 in which both the source and the destination are neighbors of y_{this}, whilst the second subprocess is concerned with the *Scenario5* in which the destination node is not a neighbor of y_{this} but the source.

Process $Interm_2$ specifies the scenarios 3 and 6 in the Figure 17. In this case the request part is similar as in $Interm_1$, however, this time there is at least one ID previous y_{this} in the ID list. Again, Rep_{interm_2} is composed of two subprocesses $Rep_{interm_2^1}$ and $Rep_{interm_2^2}$. The first one is concerned with *Scenario3* and the second process represents *Scenario6*.

Finally, process *DEST* specifies the scenario 2 in the Figure 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 modelled by the part *if* $nbr(y_{this}, y_{prv})$ *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 our 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 translation procedure is detailed in Section 9.4.

9.4 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: We refer to the first subset of rules as *protocol rules* and to the second part as *attacker rules*. The logic rules used in our method are *Horn-clauses* that are well-known in the field logic programming. In this section, we show how the logic rules are automatically derived from the protocol specification. First, we introduce the syntax of the logic rules.

9.4.1 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. We define a set $\mathcal{E}$ of mappings $\{t_1 \mapsto t_1^p\}$,..., $\{t_m \mapsto t_m^p\}$, which maps each term to the corresponding pattern.

$$t^p ::= \mathit{rreq}^p \mid \mathit{rrep}^p \mid \mathit{ID}^p \mid n^p\,[t_1^p, \ldots, t_k^p] \mid l_i^p \mid l_{src}^p, l_{dest}^p, l_{att_i}^p \mid l^p[\,] \mid x_{type}^p \mid \\ y_{this}^p, y_{prv}^p, y_{nxt}^p, y_{honprv}^p, y_{honnxt}^p, y^p \mid l_{att}^p \mid f(t_1^p, \ldots, t_k^p).$$

The patterns rreq^p, rrep^p, l_{src}^p, l_{dest}^p, $l_{att_i}^p$, and l_i^p, $i \in \{1, \ldots, k\}$, x_{type}^p represent the same things as the analogous terms in *sr*-calculus. We adapt the type system of the *sr*-calculus. y^p, y_{prv}^p, y_{nxt}^p represent variables of type node ID, thus, both the IDs of the attacker and the honest node can be bounded to them. The variables y_{this}^p, y_{honprv}^p and y_{honnxt}^p, defines the variables of type honest node's ID, hence, only l_i^p can be bounded. We add the attacker node ID type, T_{attid}, and let l_{att}^p be the variable of type T_{attid}, such that only $l_{att_i}^p$ can be bounded to l_{att}^p.

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

$$F ::= \mathit{wm}(t^p) \mid \mathit{att}(t^p) \mid \mathit{nbr}(y_{prv}^p, y_{nxt}^p) \mid \mathit{accept}([\mathit{List}]) \mid \epsilon.$$

The meaning of each fact is as follows: $\mathit{wm}(t^p)$ says that the wireless medium knows t^p, which happens when t^p is output by some node; $\mathit{att}(t^p)$ says that the attacker knows t^p, which happens when t^p is intercepted or generated by the attacker. The fact $\mathit{nbr}(y_{prv}^p, y_{nxt}^p)$ says that node y_{nxt}^p is a neighbor (within the transmission range) of node y_{prv}^p. Recall that the *nbr*-fact $\mathit{nbr}(l_{att_i}^p, l_{att_j}^p)$ is not considered during the verification because we do not assume direct links between the attacker nodes.

In the rest of the paper, we refer to these facts as *wm*-facts, *att*-facts and *nbr*-facts, respectively. The special fact $\mathit{accept}([\mathit{List}])$ is derived when the reply has got back to the source node that accepts the route specified by the ID list $[\mathit{List}]$. The fact ϵ is used to represent the successful derivation of a given fact during the deduction procedure. If the fact ϵ is reached, it means that the current derivation branch terminates successfully. The operation of routing protocols is modelled by Horn-clauses, that is, the logic rules R of form $F_1 \wedge \ldots \wedge F_n \to C$, where either F_i or C are one of the fact given above, and $\wedge$ represents a logical AND (i.e., conjuction). We 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.

Next, we detail the translation rules that are specified for translating each kind of process P in the calculus.

9.4.2 Translation rules

The automatic derivation of the protocol rules from the protocol specification in the calculus is accomplished by the translation procedure, which is based on the set of pre-defined translation rules. The set of protocol rules is defined as $[\mathcal{P}; st_{req}; st_{rep}; dir; \mathcal{H}^{req}; \mathcal{H}^{rep}; \mathcal{S}]\,\mathcal{E}$, where $\mathcal{P}$ is the process (in calculus) to be translated. $\mathcal{H}^{req}$ and $\mathcal{H}^{rep}$ are used to store the hypothesis of logic rules concerning the reply and request direction, respectively. Furthermore, $\mathcal{H}^{rep}$ is the tuple $(\mathcal{H}_{hon_1}^{rep}; \mathcal{H}_{hon_2}^{rep}; \mathcal{H}_{hon_3}^{rep}; \mathcal{H}_{att_1}^{rep}; \mathcal{H}_{att_2}^{rep})$, and $\mathcal{H}^{req}$ is the tuple $(\mathcal{H}_{hon_1}^{req}; \mathcal{H}_{hon_2}^{req}; \mathcal{H}_{hon_3}^{req}; \mathcal{H}_{hon_4}^{req}; \mathcal{H}_{hon_5}^{req}; \mathcal{H}_{hon_6}^{req}; \mathcal{H}_{att_1}^{req}; \mathcal{H}_{att_2}^{req}; \mathcal{H}_{att_3}^{req}; \mathcal{H}_{att_4}^{req})$. Each $\mathcal{H}_i^{rep}$, $\mathcal{H}_j^{req}$ represents different classes of scenarios, and is composed of logical ANDing (conjuction) of *nbr*-facts, and *wm*-facts or *att*-facts. $\mathcal{S}$ is a sequence of patterns correspond to input messages and session IDs which are used for tracking purpose. $\mathcal{E}$ is a mapping from names and variables in calculus to the corresponding names and variables in logic. σ is a set of substitutions that binds some term t to some variable x. $\mathcal{E}$ is always applied to the terms occur in $\mathcal{H}^{req}$ and $\mathcal{H}^{rep}$.

Mappings and substitutions are denoted by the mapsto arrow $\mapsto$ and leftarrow $\leftarrow$, respectively. For instance, $\{x \mapsto x^p\}$ means that variable x is mapped to pattern x^p, and $\{x \leftarrow t\}$ represents the binding of t to x. As for mappings, we note that every term has only one image in the set of patterns. st_{req} and st_{rep} are sequence of state flags, and are applied to store the current state in which the translation procedure is right before the translating process *broad* in case of the request direction, and processes *uni*, *accept* in case of the reply direction, respectively. The translation of the same process in different states yields different logic rules. Finally, *dir* is the container of flags that are used to signal the current direction in which the translation procedure is.

At the beginning, $\mathcal{E}$ includes patterns $rrep^p$, $rreq^p$, l^p_{src}, l^p_{dest}, $l^p_{att_i}$, y^p_{this}, y^p_{prv}, y^p_{nxt}, y^p_{honprv}, y^p_{honnxt}, which are the image of corresponding terms. Adding one more mapping $\{t \mapsto t^p\}$ to $\mathcal{E}$ is denoted by $\mathcal{E}\{t \mapsto t^p\}$. In case of general term t, $\mathcal{E}(t)$ is the mapping of each name and variable, which occurs in t, to the corresponding patterns. Applying the substitution σ to a term t is denoted by $t\sigma$. Adding one more fact $\mathcal{F}$ to $\mathcal{H}^{rep}_i$ and $\mathcal{H}^{req}_i$ is defined by logical ANDing $\mathcal{H}^{rep}_i$ and $\mathcal{H}^{req}_i$ with $\mathcal{F}$, respectively: $\mathcal{H}^{rep}_i \wedge \mathcal{F}$, $\mathcal{H}^{req}_i \wedge \mathcal{F}$. Adding one more logic rule R to the rule set $[\mathcal{P}; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}$ is written as $[\mathcal{P}; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E} \cup \{R\}$. Adding one more flag *flag* to st_{req} and st_{rep} is written as (st_{req},*flag*) and (st_{rep},*flag*), respectively. Finally, when we write $\forall \mathcal{H}^{rep}_i \in \mathcal{H}^{rep} : \mathcal{H}^{rep}_i \wedge \mathcal{F}$ it means $\mathcal{H}^{rep}_i \wedge \mathcal{F}$ is performed for all $\mathcal{H}^{rep}_i$ in $\mathcal{H}^{rep}$. The same is valid in case of $\forall \mathcal{H}^{req}_i \in \mathcal{H}^{req} : \mathcal{H}^{req}_i \wedge \mathcal{F}$.

Going into detail, as far as $\mathcal{H}^{rep}$ is concerned five different categories of scenarios are distinguished, which are represented by $\mathcal{H}^{rep}_{hon_1}$, $\mathcal{H}^{rep}_{hon_2}$, $\mathcal{H}^{rep}_{hon_3}$, $\mathcal{H}^{rep}_{att_1}$ and $\mathcal{H}^{rep}_{att_2}$. The first three data structures are indexed by term *hon* which refers to the scenarios in which a reply is sent by a honest node, and the last two indexed by *att* take into account the cases when the attacker forwards the reply. Additionally, $\mathcal{H}^{rep}_{hon_1}$ stores the left side of such logic rules that are concerned with the cases when a honest node received a reply, of which it is the addressee, sent by some another honest node. $\mathcal{H}^{rep}_{hon_1}$ comprises of *wm*-facts and *nbr*-facts, and it is initialized to be empty. $\mathcal{H}^{rep}_{hon_2}$ stores the left side of the rules describing the cases when the attacker is the addressee of a reply sent by some honest node. $\mathcal{H}^{rep}_{hon_2}$ is a conjunction of *wm*-facts and *nbr*-facts, and it is initialized to be empty. $\mathcal{H}^{rep}_{hon_3}$ is introduced for the cases in which the attacker is not the addressee of the reply but it overhears the reply sent by its honest neighbor. $\mathcal{H}^{rep}_{hon_3}$ is a conjunction of *wm*-facts and *nbr*-facts, and it is initialized to be $nbr(y^p_{this}, l^p_{att_i})$. $\mathcal{H}^{rep}_{att_1}$ considers the scenarios when the attacker node sends a reply to some intermediate node, and it is initialized to be $nbr(l^p_{att_i}, y^p_{this})$. Finally, $\mathcal{H}^{rep}_{att_2}$ is in connection with the cases when the attacker node sends a reply to the source node, and it is initialized to be $nbr(l^p_{att_i}, l^p_{src})$. Both $\mathcal{H}^{rep}_{att_1}$ and $\mathcal{H}^{rep}_{att_2}$ are built up by logical ANDing of *att*-facts and *nbr*-facts.

While $\mathcal{H}^{rep}$ concerns the reply direction, $\mathcal{H}^{req}$ is used for specifying the scenarios concerning the request direction. Again, the possible scenarios are categorized into nine different classes, which are represented by $\mathcal{H}^{req}_{hon_i}$ and $\mathcal{H}^{req}_{att_j}$, where $1 \leq i \leq 6$ and $1 \leq j \leq 3$. In the same lines with the reply direction, the indices *hon* and *att* refer to the cases when a request is sent by a honest node and attacker node, respectively. $\mathcal{H}^{req}_{hon_1}$ considers the scenario when an intermediate honest node receives a request broadcast by the source node, and it is initialized to $nbr(l^p_{src}, y^p_{this})$. $\mathcal{H}^{req}_{hon_2}$ is the left side of such logic rules that describe the case when the attacker intercepts the request broadcast by the source node, and it is initialized to $nbr(l^p_{src}, l^p_{att_i})$. $\mathcal{H}^{req}_{hon_3}$ is related to the scenario in which a honest intermediate node receives a request output by an another honest intermediate node. $\mathcal{H}^{req}_{hon_3}$ is initialized to $nbr(y^p_{honprv}, y^p_{this})$. $\mathcal{H}^{req}_{hon_4}$ is $nbr(y^p_{this}, l^p_{att_i})$ at the beginning, and concerns the case in which the attacker obtains the request sent by an intermediate honest node. $\mathcal{H}^{req}_{hon_5}$ and $\mathcal{H}^{req}_{hon_6}$ are related to the process *DEST* and represent the hypothesis of the rules from the destination's point of view. $\mathcal{H}^{req}_{hon_5}$ considers the case when the destination receives a request broadcast by an intermediate honest node and the attacker is not a neighbor of the destination. $\mathcal{H}^{req}_{hon_6}$ is similar to $\mathcal{H}^{req}_{hon_5}$ but in this case the attacker is in the destination's range. $\mathcal{H}^{req}_{hon_5}$ is initialized to $nbr(y^p_{honprv}, l^p_{dest})$, whilst at the beginning $\mathcal{H}^{req}_{hon_6}$ is $nbr(y^p_{honprv}, l^p_{dest}) \wedge nbr(l^p_{dest}, l^p_{att_i})$. Finally, $\mathcal{H}^{req}_{att_1}$, $\mathcal{H}^{req}_{att_2}$ and $\mathcal{H}^{req}_{att_3}$ contains the hypothesis of such logic rules that describe the cases when the attacker node sends a request. They are in turn initialized to $nbr(l^p_{att_i}, y^p_{this})$, $nbr(l^p_{att_i}, l^p_{dest})$, and $nbr(l^p_{att_i}, l^p_{dest}) \wedge nbr(l^p_{dest}, l^p_{att_i})$.

st_{req} and st_{rep} are local variables and are containers of a sequence of constant flags *HONEST*, *ATTPRV*, and *ATTNXT*. *dir* is a local variable and is composed of constant flags *REQ* and *REP*. At the beginning, *dir* is initialized to *REQ* and it is changed to *REP* whenever the translation reaches the process concerning the reply direction. Both st_{req} and st_{rep} are initialized to be empty. They are updated (not at the same time) whenever a process of form $letor\ (y_{xxx} = y_{honxxx})\ or\ (y_{xxx} = l_{att_i})\ in\ P$ has been translated, where *xxx* can be *prv* or *nxt*. st_{req} is updated when the value of *dir* is *REQ*, whilst st_{rep} is modified when the value of *dir* is *REP*. The part $(y_{xxx} = x_{honxxx})$ says that x is bound to an ID of honest node and yields adding *HONEST*, while $(y_{honprv} = l_{att_i})$ and $(y_{honnxt} = l_{att_i})$ binds l_{att_i} to the variables, and the flags *ATTPRV*, *ATTNXT* are added to st_{req} and st_{rep}, respectively. In our method the following translation rules are specified:

T_1. $[\mathbf{0}; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E} = \emptyset$

T_2. $[P|Q; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E} = [P; st_{req}; st_{rep}; dir; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}\ \cup$
$[Q; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}$

T_3. $[!P; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E} = [P; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; (\mathcal{S}, s)]\, \mathcal{E}\{s \mapsto s^p\}$,
where s is a new variable session identifier

T_4. $[(new\ n)P; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E} = [P; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}\{n \mapsto n^p[\mathcal{S}]\}$

T_5. $[let\ (y_{this} = nid)\ in\ P; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E} =$
$[P\{y_{this} \leftarrow nid\}; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}$

T^1_6. $[letor\ (y_{prv} = y_{honprv})\ or\ (y_{prv} = l_{att_i})\ in\ P; st_{req}; st_{rep}; REQ; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E} =$
$[P\{y_{prv} \leftarrow y_{honprv}\}; (st_{req}, \text{HONEST}); st_{rep}; REQ; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}\ \cup$
$[P\{y_{prv} \leftarrow l_{att_i}\}; (st_{req}, \text{ATTPRV}); st_{rep}; REQ; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}$

T^2_6. $[letor\ (y_{prv} = y_{honprv})\ or\ (y_{prv} = l_{att_i})\ in\ P; st_{req}; st_{rep}; REP; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E} =$
$[P\{y_{prv} \leftarrow y_{honprv}\}; st_{req}; (st_{rep}, \text{HONEST}); REP; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}\ \cup$
$[P\{y_{prv} \leftarrow l_{att_i}\}; st_{req}; (st_{rep}, \text{ATTPRV}); REP; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}$

T^3_6. $[letor\ (y_{nxt} = y_{honnxt})\ or\ (y_{nxt} = l_{att_i})\ in\ P; st_{req}; st_{rep}; REP; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E} =$
$[P\{y_{nxt} \leftarrow y_{honnxt}\}; st_{req}; (st_{rep}, \text{HONEST}); REP; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}\ \cup$
$[P\{y_{nxt} \leftarrow l_{att_i}\}; st_{req}; (st_{rep}, \text{ATTNXT}); REP; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}$

T_7. $[recvreq(t_1).P; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; (\mathcal{S}, t_1)]\, \mathcal{E} =$
$[P;\ st_{req};\ st_{rep};\ dir;\ \mathcal{H}^{rep};\ \mathcal{H}^{req}_{hon_i} \wedge wm(t_1);\ \mathcal{H}^{req}_{att_j} \wedge att(t_1)$ for every value of i and j, where $1 \leq i \leq 6$ and $1 \leq j \leq 3$; $(\mathcal{S}, t_1)]\mathcal{E}\{t_1 \mapsto t^p_1\}$

T^1_8. $[broad\,(t_2)\,.P; empty; st_{rep}; REQ; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle \mathcal{S}, t_1 \rangle]\, \mathcal{E} =$
$[P; empty; st_{rep}; REP; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle \mathcal{S}, t_1 \rangle]\, \mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{\mathcal{H}^{req}_{hon_1} \rightarrow wm(t_2)\}$
$\cup\ \{\mathcal{H}^{req}_{hon_2} \rightarrow att(t_1)\} \cup \{\mathcal{H}^{req}_{att_1} \rightarrow wm(t_2)\}$

T^2_8. $[broad\,(t_2)\,.P; (\text{HONEST}); st_{rep}; REQ; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E} =$
$[P; (\text{HONEST}); st_{rep}; REP; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{\mathcal{H}^{req}_{hon_3} \rightarrow wm(t_2)\}$
$\cup\ \{\mathcal{H}^{req}_{att_1} \rightarrow wm(t_2)\}$

T^3_8. $[broad\,(t_2)\,.P; (\text{ATTPRV}); st_{rep}; REQ; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle \mathcal{S}, t_1 \rangle]\, \mathcal{E} =$
$[P; (\text{ATTPRV}); st_{rep}; REP; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle \mathcal{S}, t_1 \rangle]\, \mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{\mathcal{H}^{req}_{hon_4} \rightarrow att(t_1)\}$
$\cup\ \{\mathcal{H}^{req}_{att_1} \rightarrow wm(t_2)\}$

T_9. $[broadinit\,(t_2)\,.P; st_{req}; st_{rep}; REQ; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E} =$
$[P; st_{req}; st_{rep}; REP; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\, \mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{wm(t_2)\}$

T_{10}. $[recvrep(t_1).P; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E} =$
$[P;\ st_{req};\ st_{rep};\ dir;\ \mathcal{H}^{rep}_{hon_1} \wedge wm(t_1);\ \mathcal{H}^{rep}_{hon_2} \wedge wm(t_1);\ \mathcal{H}^{rep}_{hon_3} \wedge wm(t_1);\ \mathcal{H}^{rep}_{att_1} \wedge att(t_1);$
$\mathcal{H}^{rep}_{att_2} \wedge att(t_1);\ \mathcal{H}^{req};\ (\mathcal{S},\ t_1)]\ \mathcal{E}\{t_1 \mapsto t^p_1\}$

T^1_{11}. $[uni\,(t_2)\,.P; st_{req}; empty; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E} =$
$[P; st_{req}; empty; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{wm(t_1) \wedge \mathcal{H}^{rep}_{hon_1} \rightarrow wm(t_2)\} \cup$
$\{wm(t_1) \wedge \mathcal{H}^{rep}_{hon_3} \rightarrow att(t_2)\} \cup \{wm(t_1) \wedge \mathcal{H}^{rep}_{att_1} \rightarrow wm(t_2)\}$

T^2_{11}. $[uni\,(t_2)\,.P; st_{req}; (\text{HONEST}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E} =$
$[P; st_{req}; (\text{HONEST}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{wm(t_1) \wedge \mathcal{H}^{rep}_{hon_1} \rightarrow wm(t_2)\} \cup$
$\{wm(t_1) \wedge \mathcal{H}^{rep}_{hon_3} \rightarrow att(t_2)\} \cup \{wm(t_1) \wedge \mathcal{H}^{rep}_{att_1} \rightarrow wm(t_2)\}$

T^3_{11}. $[uni\,(t_2)\,.P; st_{req}; (\text{ATTPRV}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E} =$
$[P; st_{req}; (\text{ATTPRV}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{wm(t_1) \wedge \mathcal{H}^{rep}_{hon_2} \rightarrow att(t_2)\}$

T^4_{11}. $[uni\,(t_2)\,.P; st_{req}; (\text{ATTNXT}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E} =$
$[P; st_{req}; (\text{ATTNXT}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{wm(t_1) \wedge \mathcal{H}^{rep}_{att_1} \rightarrow wm(t_2)\}$

T^5_{11}. $[uni\,(t_2)\,.P; st_{req}; (\text{HONEST}, \text{HONEST}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E} =$
$[P; st_{req}; (\text{HONEST}, \text{HONEST}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{wm(t_1) \wedge \mathcal{H}^{rep}_{hon_1} \rightarrow wm(t_2)\}$
$\cup\ \{wm(t_1) \wedge \mathcal{H}^{rep}_{hon_3} \rightarrow att(t_2)\} \cup \{wm(t_1) \wedge \mathcal{H}^{rep}_{att_1} \rightarrow wm(t_2)\}$

T^6_{11}. $[uni\,(t_2)\,.P; st_{req}; (\text{ATTPRV}, \text{HONEST}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E} =$
$[P; st_{req}; (\text{ATTPRV}, \text{HONEST}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{wm(t_1) \wedge \mathcal{H}^{rep}_{hon_2} \rightarrow att(t_2)\}$

T^7_{11}. $[uni\,(t_2)\,.P; st_{req}; (\text{ATTNXT}, \text{HONEST}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E} =$
$[P; st_{req}; (\text{ATTNXT}, \text{HONEST}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \langle\mathcal{S}, t_1\rangle]\,\mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{wm(t_1) \wedge \mathcal{H}^{rep}_{att_1} \rightarrow wm(t_2)\}$

T^1_{12}. $[accept(t_3); st_{req}; (\text{HONEST}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E} =$
$[\mathbf{0}; st_{req}; (\text{HONEST}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E}\{t_3 \mapsto t^p_3\} \cup \{\mathcal{H}^{rep}_{hon_1} \rightarrow accept(t_3)\} \cup$
$\{\mathcal{H}^{rep}_{att_2} \rightarrow accept(t_3)\}$

T^2_{12}. $[accept(t_3); st_{req}; (\text{ATTNXT}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E} =$
$[\mathbf{0}; st_{req}; (\text{ATTNXT}); dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E}\{t_3 \mapsto t^p_3\} \cup \{\mathcal{H}^{rep}_{att_2} \rightarrow accept(t_3)\}$

T_{13}. $[if\ \ nbr(y_k, y_t)\ \ then\ P; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E} =$
$[P;\ st_{req};\ st_{rep};\ dir;\ \forall\ \mathcal{H}^{rep}_i \in \mathcal{H}^{rep}\colon\ \mathcal{H}^{rep}_i \wedge nbr(y_k, y_t);\ \forall\ \mathcal{H}^{req}_j \in \mathcal{H}^{req}/\{\mathcal{H}^{req}_{hon_2}, \mathcal{H}^{req}_{hon_4}\}\colon$
$\mathcal{H}^{req}_j \wedge nbr(y_k, y_t);\ \mathcal{S}]\ \mathcal{E}\{y_k \mapsto y^p_k\}\{y_t \mapsto y^p_t\}$

T_{14}. $[letdst\ x = g(t_1, \ldots, t_n)\ in\ P; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E} =$
$\bigcup\ \{[P; st_{req}; st_{rep}; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\mathcal{E}\{x \mapsto \sigma_2 p\}\sigma\sigma_1$
where $g(p_1, \ldots, p_n) = p$, *and* (σ_1, σ_2) *is a most general unifier such that*
$\sigma_1\mathcal{E}(t_i) = \sigma_2 p_i,\ 1 \leq i \leq n\}$

T^1_{15}. $[unidest\,(t_2)\,.P; (\text{HONEST}); st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E} =$
$[P; (\text{HONEST}); st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{\mathcal{H}^{req}_{hon_5} \rightarrow wm(t_2)\}$
$\cup\ \{\mathcal{H}^{req}_{hon_6} \rightarrow att(t_2)\} \cup \{\mathcal{H}^{req}_{att_2} \rightarrow att(t_2)\} \cup \{\mathcal{H}^{req}_{att_3} \rightarrow wm(t_2)\}$

T^2_{15}. $[unidest\,(t_2)\,.P; (\text{ATTPRV}); st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E} =$
$[P; (\text{ATTPRV}); st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E}\{t_2 \mapsto t^p_2\} \cup \{\mathcal{H}^{req}_{att_3} \rightarrow att(t_2)\}$

The translation procedure can be seen as a consecutive execution of translation rules, which follows the next processing precedence: By default, going from the left to the right, the keyword of each processes are looked for. In case of $P|Q$, the parallel composition has the highest precedence. However, this is not the case when boundaries are used: In $(new\ ID)(P|Q)$, $let \ldots in\ (P|Q)$, and $letor \ldots in\ (P|Q)$ the parts $(new\ ID)$, $let \ldots in$, and $letor \ldots in$ have higher precedence than parallel composition.

Rule T_1 says that the translation of the nil process in any state and direction yields no rule. T_2 says that the logic rules used for specifying process $P|Q$ is the union of the rules correspond to P and rules correspond to Q. T_3 means that the logic rules for describing process $!P$ is the same as the rules for P but a new session ID is created and stored, which is rendered to a particular instance of P. The notion of replication is not explicitly expressed but it is involved implicitly in such a way that each logic rule corresponding to P can take place several times in different resolution steps. Rule T_4 expresses that the logic rules of $(new\ n)P$ is the same as P, in which every occurence of n in P is translated to $n^p[\,]$, this is defined by adding new mapping $\{n \mapsto n^p[\mathcal{S}]\}$ to $\mathcal{E}$.

T_5 is concerned with the translation of process $let\ (y_{this} = nid)\ in\ P$, where id can be l_{src}, l_{dest}, l_{att_i}, y_{honprv}, y_{honnxt}. The logic rules for specifying $let\ (y_{this} = nid)\ in\ P$ is the same as the logic rules for $P\{y_{this} \leftarrow nid\}$. As for T_6, three different subrules are distinguished regarding the value of the states and direction. The first rule considers the REQ direction whilst the other two rules taking into account the REP direction. The translation of process $letor\ (y_{xxx} = y_{honxxx})\ or\ (y_{xxx} = l_{att_i})\ in\ P$ can be seen as the union of the logic rules for specifying $P\{y_{xxx} \leftarrow y_{honxxx}\}$ and the logic rules for specifying $P\{y_{xxx} \leftarrow l_{att_i}\}$. In case of $P\{y_{xxx} \leftarrow y_{honxxx}\}$ the flag HONEST is added to st_{req} and st_{rep}, depeding on the current direction. In the branch $P\{y_{xxx} \leftarrow l_{att_i}\}$ the flags ATTPRV and ATTNXT are added when xxx is prv and nxt, respectively.

T_7 is invoked when a process to be translated is $recvreq(t_1).P$. $recvreq(t_1)$ is eiminated and $\mathcal{H}^{req}_{hon_i}$ is extended with $\wedge\ wm(t_1)$ while $\mathcal{H}^{req}_{att_j}$ is extended with $\wedge\ att(t_1)$. Finally, $\mathcal{E}$ is updated with the mapping $\{t_1 \mapsto t_1^p\}$ and $\mathcal{S}$ is extended with a the input message t_1^p. Rules T_8^1, T_8^2 and T_8^3 are used for translating process $broad\,(t_2)\,.P$, and are invoked when its st_{req} is empty, HONEST and ATTPRV, respectively. As the result, the procedure in all the three cases carries on with translating P in the direction REP. In addition, the rule sets are updated with new logic rules. We note that $\langle \mathcal{S}, t_1 \rangle$ in T_8^1 and T_8^2 says that the last input message in $\mathcal{S}$ is t_1.

T_9 is very similar to T_8, the only difference is that it is defined from the source's point of view, and only $wm(t_2^p)$ is added. Note that $wm(t_2^p)$ can be seen as the rule of which the hypothesis is empty. Rule T_{10} translates process $recvrep(t_1).P$ in the very similar way as in case of T_7 apart from that the current value of the state and direction is off the point; Rule T_{11} is responsible for handling process $uni\,(t_2)\,.P$, and is classified into seven subrules according to the value of st_{rep}.

Whenever the translation procedure gets to the point where $accept(t_3)$ is to be translated, rules T^1_{12} and T^2_{12} are invoked. The first rule is valid when the first ID of the ID list included in the reply belongs to honest nodes, and the second rule holds when the first ID is l_{att_i}. T_{13} extends all H_j^{rep} and H_i^{req} with $\wedge\ nbr(y_k, y_t)$, except for $H^{req}_{hon_2}$ and $H^{req}_{hon_4}$. Finally, rules T^1_{15}, T^2_{15} are used for translating process $unidest\,(t_2)\,.P$ that is similar to T_{11} except that other rules are added to the rule set.

At the end, the translation procedure results in a set $\mathcal{L}$ of logic rules. As the subsequent step, the rules in $\mathcal{L}$ are sanitized in which duplicated nbr-facts in the hypothesis are eliminated, and the facts in each hypothesis are reordered, such that the wm-facts and att-facts are placed on the leftmost position.

9.4.3 The resulting protocol rules

The translation $[P_{spec}; st_{req}; st_{rep}; dir; \mathcal{H}^{rep}; \mathcal{H}^{req}; \mathcal{S}]\,\mathcal{E}$ of the protocol specification P_{spec} given in Section 9.3 yields the following logic rules (after sanitizing).

(Protocol rules: Template of the correct operation) ::=
R_1^{req}. $wm(t^p_{req})$.

$R_{2.1}^{req}$. $wm(t^p_{req}) \wedge nbr(l^p_{src}, y^p_{this}) \wedge nbr(y^p_{this}, l^p_{src}) \wedge Prot^{facts}_{intermreq} \rightarrow wm(t^p_{req'})$.
$R_{2.2}^{req}$. $wm(t^p_{req}) \wedge nbr(l^p_{src}, l^p_{att_i}) \rightarrow att(t^p_{req})$.
$R_{2.3}^{req}$. $wm(t^p_{req}) \wedge nbr(y^p_{honprv}, y^p_{this}) \wedge nbr(y^p_{this}, y^p_{honprv}) \wedge Prot^{facts}_{intermreq} \rightarrow wm(t^p_{req'})$.
$R_{2.4}^{req}$. $wm(t^p_{req}) \wedge nbr(y^p_{honprv}, l^p_{att_i}) \rightarrow att(t^p_{req'})$.
$R_{2.5}^{req}$. $att(t^p_{req}) \wedge nbr(l^p_{att_i}, y^p_{this}) \wedge nbr(y^p_{this}, l^p_{att_i}) \rightarrow att(t^p_{req'})$.

$R^{req}_{2.6}$. $att(t^p_{req}) \wedge nbr(l^p_{att_i}, y^p_{this}) \wedge nbr(y^p_{this}, y^p_{honprv}) \wedge Prot^{facts}_{intermreq} \to wm(t^p_{req'})$.
$R^{req}_{2.7}$. $att(t^p_{req}) \wedge nbr(l^p_{att_i}, y^p_{this}) \wedge nbr(y^p_{this}, l^p_{src}) \wedge Prot^{facts}_{intermreq} \to wm(t^p_{req'})$.

$R^{req}_{3.1}$. $wm(t^p_{req}) \wedge nbr(y^p_{honprv}, l^p_{dest}) \wedge nbr(l^p_{dest}, y^p_{honprv}) \wedge Prot^{facts}_{dest} \to wm((y^p_{honprv}, t^p_{rep}))$.
$R^{req}_{3.2}$. $wm(t^p_{req}) \wedge nbr(y^p_{honprv}, l^p_{dest}) \wedge nbr(l^p_{dest}, y^p_{honprv}) \wedge nbr(l^p_{dest}, l^p_{att_i}) \wedge$
$Prot^{facts}_{dest} \to att((y^p_{honprv}, t^p_{rep}))$.
$R^{req}_{3.3}$. $att(t^p_{req}) \wedge nbr(l^p_{att_i}, l^p_{dest}) \wedge nbr(l^p_{dest}, l^p_{att_i}) \wedge nbr(l^p_{dest}, y^p_{honprv}) \wedge$
$Prot^{facts}_{dest} \to att((y^p_{honprv}, t^p_{rep}))$.
$R^{req}_{3.4}$. $att(t^p_{req}) \wedge nbr(l^p_{att_i}, l^p_{dest}) \wedge nbr(l^p_{dest}, l^p_{att_i}) \wedge Prot^{facts}_{dest} \to att((l^p_{att_i}, t^p_{rep}))$.
$R^{req}_{3.5}$. $att(t^p_{req}) \wedge nbr(l^p_{att_i}, l^p_{dest}) \wedge nbr(l^p_{dest}, y^p_{honprv}) \wedge Prot^{facts}_{dest} \to wm((y^p_{honprv}, t^p_{rep}))$.

$R^{rep}_{1.1}$. $wm(t^p_{req}) \wedge wm((l^p_{src}, t^p_{rep})) \wedge nbr(l^p_{src}, y^p_{honnxt})$
$\wedge\ Prot^{facts}_{init} \to accept([y^p_{honnxt}, List])$.
$R^{rep}_{1.2}$. $wm(t^p_{req}) \wedge att((l^p_{src}, t^p_{rep})) \wedge nbr(l^p_{src}, l^p_{att_i})$
$\wedge\ nbr(l^p_{att_i}, l^p_{src}) \wedge Prot^{facts}_{init} \to accept([l^p_{att_i}, List])$.
$R^{rep}_{1.3}$. $wm(t^p_{req}) \wedge att((l^p_{src}, t^p_{rep})) \wedge nbr(l^p_{src}, y^p_{honnxt})$
$\wedge\ nbr(l^p_{att_i}, l^p_{src}) \wedge Prot^{facts}_{init} \to accept([y^p_{honnxt}, List])$.

$R^{rep}_{2.1}$. $RequestPart \wedge wm((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, l^p_{dest}) \wedge nbr(y^p_{this}, l^p_{src})$
$\wedge\ Prot^{facts}_{intermrep} \to wm((l^p_{src}, t^p_{rep'}))$.

$R^{rep}_{2.2}$. $RequestPart \wedge wm((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, l^p_{dest}) \wedge nbr(y^p_{this}, l^p_{src})$
$\wedge\ nbr(y^p_{this}, l^p_{att_i}) \wedge Prot^{facts}_{intermrep} \to att((l^p_{src}, t^p_{rep'}))$.

$R^{rep}_{2.3}$. $RequestPart \wedge att((y^p_{this}, t^p_{rep})) \wedge nbr(l^p_{att_i}, y^p_{this}) \wedge nbr(y^p_{this}, l^p_{dest})$
$\wedge\ nbr(y^p_{this}, l^p_{src}) \wedge Prot^{facts}_{intermrep} \to wm((l^p_{src}, t^p_{rep'}))$.

$R^{rep}_{2.4}$. $RequestPart \wedge wm((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, y^p_{honnxt})$
$\wedge\ nbr(y^p_{this}, y^p_{honnxt}) \wedge Prot^{facts}_{intermrep} \to wm((l^p_{src}, t^p_{rep'}))$.

$R^{rep}_{2.5}$. $RequestPart \wedge wm((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, y^p_{honnxt}) \wedge nbr(y^p_{this}, y^p_{honnxt})$
$\wedge\ nbr(y^p_{this}, l^p_{att_i}) \wedge Prot^{facts}_{intermrep} \to att((l^p_{src}, t^p_{rep'}))$.

$R^{rep}_{2.6}$. $RequestPart \wedge att((y^p_{this}, t^p_{rep})) \wedge nbr(l^p_{att_i}, y^p_{this}) \wedge nbr(y^p_{this}, l^p_{honnxt})$
$\wedge\ nbr(y^p_{this}, l^p_{src}) \wedge Prot^{facts}_{intermrep} \to wm((l^p_{src}, t^p_{rep'}))$.

$R^{rep}_{2.7}$. $RequestPart \wedge att((y^p_{this}, t^p_{rep})) \wedge nbr(l^p_{att_i}, y^p_{this}) \wedge nbr(y^p_{this}, l^p_{att_i})$
$\wedge\ nbr(y^p_{this}, l^p_{src}) \wedge Prot^{facts}_{intermrep} \to wm((l^p_{src}, t^p_{rep'}))$.

$R^{rep}_{2.8}$. $RequestPart \wedge wm((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, l^p_{dest}) \wedge nbr(y^p_{this}, y^p_{honprv})$
$\wedge\ Prot^{facts}_{intermrep} \to wm((y^p_{honprv}, t^p_{rep'}))$.

$R^{rep}_{2.9}$. $RequestPart \wedge wm((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, l^p_{dest}) \wedge nbr(y^p_{this}, y^p_{honprv})$
$\wedge\ nbr(y^p_{this}, l^p_{att_i}) \wedge Prot^{facts}_{intermrep} \to att((y^p_{honprv}, t^p_{rep'}))$.

$R^{rep}_{2.10}$. $RequestPart \wedge att((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, l^p_{dest}) \wedge nbr(y^p_{this}, y^p_{honprv})$
$\wedge\ nbr(l^p_{att_i}, y^p_{this}) \wedge Prot^{facts}_{intermrep} \to wm((y^p_{honprv}, t^p_{rep'}))$.

$R^{rep}_{2.11}$. $RequestPart \wedge wm((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, l^p_{dest}) \wedge nbr(y^p_{this}, l^p_{att_i})$
$\wedge\ Prot^{facts}_{intermrep} \to att((l^p_{att_i}, t^p_{rep'}))$.

$R^{rep}_{2.12}$. $RequestPart \wedge wm((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, y^p_{honnxt}) \wedge nbr(y^p_{this}, y^p_{honprv})$
$\wedge\ Prot^{facts}_{intermrep} \to wm((y^p_{honprv}, t^p_{rep'}))$.

$R^{rep}_{2.13}$. $RequestPart \wedge wm((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, y^p_{honnxt}) \wedge nbr(y^p_{this}, y^p_{honprv})$
$\wedge\, nbr(y^p_{this}, l^p_{att_i}) \wedge Prot^{facts}_{intermrep} \rightarrow att((y^p_{honprv}, t^p_{rep'}))$.

$R^{rep}_{2.14}$. $RequestPart \wedge att((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, y^p_{honnxt}) \wedge nbr(y^p_{this}, y^p_{honprv})$
$\wedge\, nbr(l^p_{att_i}, y^p_{this}) \wedge Prot^{facts}_{intermrep} \rightarrow wm((y^p_{honprv}, t^p_{rep'}))$.

$R^{rep}_{2.15}$. $RequestPart \wedge wm((y^p_{this}, t^p_{rep})) \wedge nbr(y^p_{this}, y^p_{honnxt}) \wedge nbr(y^p_{this}, l^p_{att_i})$
$\wedge\, Prot^{facts}_{intermrep} \rightarrow att((l^p_{att_i}, t^p_{rep'}))$.

$R^{rep}_{2.16}$. $RequestPart \wedge att((y^p_{this}, t^p_{rep})) \wedge nbr(l^p_{att_i}, y^p_{this}) \wedge nbr(y^p_{this}, l^p_{att_i})$
$\wedge\, nbr(y^p_{this}, y^p_{honprv}) \wedge Prot^{facts}_{intermrep} \rightarrow wm((y^p_{honprv}, t^p_{rep'}))$.

Again we note that these rules consider a general case, the protocol dependent processing parts, and the request and reply messages t^i_{req} and t^j_{rep} are defined by a specific protocol. Figure 18, Figure 19 and Figure 20 illustrate the scenarios described by each logic rule. In the figures, the nodes labelled by l_{src}, l_{dest} and l_{att} are the source, the destination and the attacker nodes. The remaining nodes represent honest intermediate nodes. The arrow with the label req or 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 req or rep from node A to node B means that B has received the request or reply sent by A. The arrow with label nbr represents the neighbor check performed by a node after receiving some request or reply. For instance, the edge labelled by nbr from l^p_{src} to l^p_{att} means that the source node checks whether the attacker is its neighbor. The arrow with label (nbr, req) or (nbr, 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 we provide the interpretation of some scenarios in the Figures: In $R^{req}_{2.1}$, after receiving a request broadcast by the source, y_{this} checks if l_{src} is its neighbor, if so, it re-broadcasts the request. $R^{req}_{2.7}$ present the case when the attacker impersonates l_{src}, and says that after y_{this} receives the request sent by l_{att} it checks if the source is its neighbor. Finally, $R^{rep}_{2.2}$ says that after performing neighbor checks, y_{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.

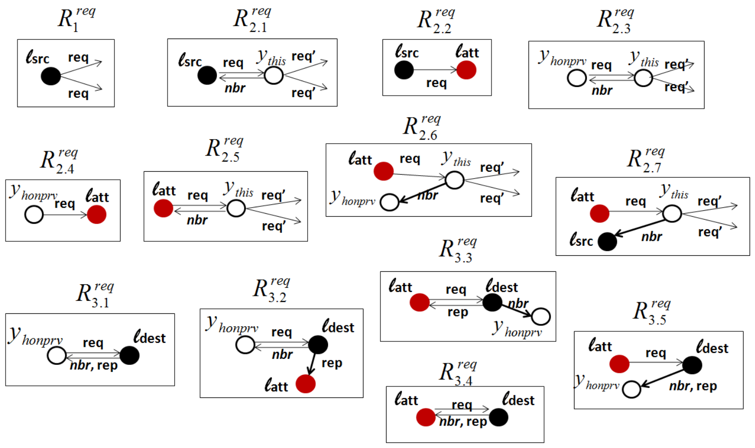
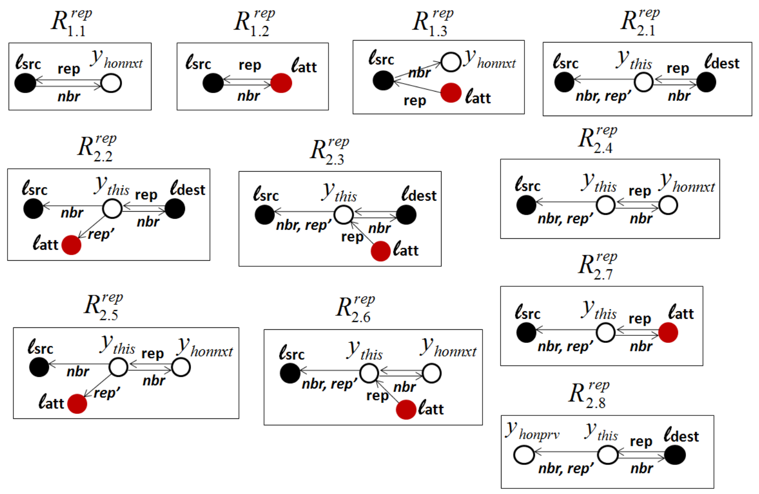


Figure 18: The scenarios corredponding to the logic rules which are concerned with the request direction.

Rule R^{req}_1 and $R^{rep}_{1.1}$, $R^{rep}_{1.2}$, $R^{rep}_{1.3}$ model the operation of the source node. R^{req}_1 says that the initial request has been broadcast by the source node. Rule $R^{rep}_{1.1}$ describes the case when l_{src} receives a reply sent by a honest node, rules $R^{rep}_{1.2}$ specifies the scenario when the source node receives a reply sent by the attacker node and the first ID in the ID list is l_{att}. Rule $R^{rep}_{1.3}$ is the same as $R^{rep}_{1.2}$ except for the first ID in the ID list is some honest node y_{honnxt}, which is concerned with the case when the attacker impersonates y_{honnxt}.

Rules $R^{req}_{2.i}$ and $R^{rep}_{2.j}$ are concerned with the operation of intermediate nodes. $R^{req}_{2.1}$ says that when y_{this} obtains a request broadcast by l_{src}, first, it checks if l_{src} is its neighbor, thereafter, the request is processed and re-broadcast. $R^{req}_{2.2}$ specifies the scenario in which the attacker receives the initial request. Rules $R^{req}_{2.3}$ and $R^{req}_{2.4}$ are similar to the previous two rules, but this time the sender is some honest intermediate node y_{honprv}. Rules $R^{req}_{2.5}$ and $R^{req}_{2.6}$ 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}. $R^{req}_{2.7}$ is similar to $R^{req}_{2.6}$ except that in $R^{req}_{2.7}$ the attacker impersonates the source node.



$R^{rep}_{1.1}$
$l$src rep y_{honnxt} nbr
$R^{rep}_{1.2}$
$l$src rep $l$att nbr
$R^{rep}_{1.3}$
$l$src nbr y_{honnxt} rep $l$att
$R^{rep}_{2.1}$
$l$src y_{this} rep $l$dest nbr, rep' nbr
$R^{rep}_{2.2}$
$l$src y_{this} rep $l$dest nbr nbr $l$att rep'
$R^{rep}_{2.3}$
$l$src y_{this} $l$dest nbr, rep' rep nbr $l$att
$R^{rep}_{2.4}$
$l$src y_{this} rep y_{honnxt} nbr, rep' nbr
$R^{rep}_{2.5}$
$l$src y_{this} rep y_{honnxt} nbr nbr $l$att rep'
$R^{rep}_{2.6}$
$l$src y_{this} y_{honnxt} nbr, rep' rep nbr $l$att
$R^{rep}_{2.7}$
$l$src y_{this} rep $l$att nbr, rep' nbr
$R^{rep}_{2.8}$
y_{honprv} y_{this} rep $l$dest nbr, rep' nbr

Figure 19: The scenarios corredponding to the logic rules which are concerned with the reply direction, part 1.

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

The rules $R^{rep}_{2.1}, \ldots, R^{rep}_{2.16}$ 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 $R^{rep}_{2.1}$, $R^{rep}_{2.2}$, and $R^{rep}_{2.3}$, 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 $R^{rep}_{2.4} - R^{rep}_{2.6}$ are the same to the previous three rules with l_{dest} is replaced by some honest intermediate node y_{honnxt}. $R^{rep}_{2.7}$ and $R^{rep}_{2.16}$ describes the case when ID following y_{this} is l_{att}. Rules $R^{rep}_{2.8} - R^{rep}_{2.10}$ are the same to the rules $R^{rep}_{2.4} - R^{rep}_{2.6}$ but it involves an intermediate node instead of l_{src}. Rules $R^{rep}_{2.12} - R^{rep}_{2.14}$ involves only intermediate nodes. Eventually, in $R^{rep}_{2.11}$ and $R^{rep}_{2.15}$ the attacker node is the addressee of the reply.

RequestPart in each reply rule considers the corresponding request of the reply, such that when a reply is received by some node, it is accepted only in case its corresponding request has been broadcast before. The part $Prot^{facts}$ in the rules corresponds to the processes $Prot$ and $Prot^{Rep}$ in the calculus specification. They represent additional protocol specific processing steps, which usually is an additional neighbor check or an execution of some function such as signature check, decryption, computing hash and MAC values. One advantage of our method is that it is based resolution which includes the application of unification methods. Unification can be used

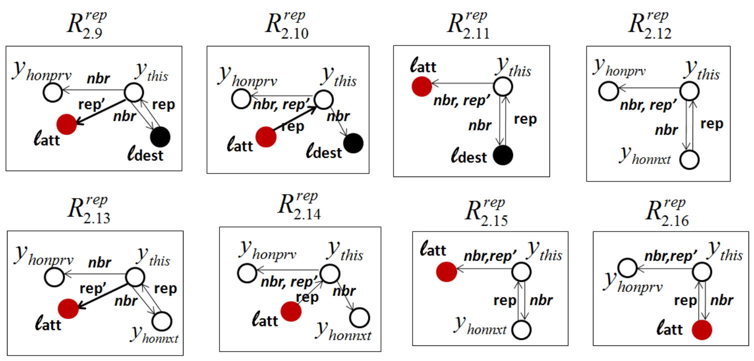


Figure 20: The scenarios corredponding to the logic rules which are concerned with the reply direction, part 2.

to model implicit the verification procedure such as MAC, hash and signature verification. More details about unification and resolution can be found in Section 9.5. Therefore only additional neighbor checks should be explicitly defined, which means that $Prot^{facts}$ can either be composed of *nbr*-facts or can be empty.

9.4.4 Specifying the attacker rules

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

$(Init.\ knowl.) ::= \forall l^p_i$ neighbors of l^p_{att}:
$I^{ownID}_{att}.\ att(l^p_{att}),\ I^{nbrID}_{att}.\ nbr(l^p_{att}, l^p_i) \rightarrow att(l^p_i);\ I^{sKey}_{att}.\ att(k(l_{att}, l^p_i));$
$I^{pKey}_{att}.\ att(pk(l^p_j))$ for all honest l^p_j;

I^{ownID}_{att} and I^{nbrID}_{att} mean that initially the attacker knows its own ID and the IDs of its honest neighbors, respectively. I^{sKey}_{att} and I^{pKey}_{att} say that the attacker possesses all the keys it shares with the honest nodes, and all public keys. Function $pk(l^p_j)$ represents the public key of node l^p_j. We denote the set of knowledge of the attacker as $\mathcal{K}_{att}$. At the beginning, $\mathcal{K}_{att} = I^{ownID}_{att} \cup I^{nbrID}_{att}$ $I^{sKey}_{att} \cup I^{pKey}_{att}$. In addition, we define a computation ability for the attacker node as follows:

(Computation ability - protocol independent) ::=
$A^{data}.\ att(s) \rightarrow att(n^p[s])$

$A^{fun}.$ *For each public function f of n-arity*
$att(x^p_1) \wedge \cdots \wedge att(x^p_n) \rightarrow att(f(x^p_1, \ldots, x^p_n));$
$A^{hash}.\ att(x^p) \rightarrow att(h(x^p));$
$A^{senc}.\ att(x^p) \wedge att(k^p(y^p_{nid_i}, y^p_{nid_j})) \rightarrow att(senc(x^p, k^p(y^p_{nid_i}, y^p_{nid_j})));$
$A^{penc}.\ att(x^p) \wedge att(pk(y^p_{nid})) \rightarrow att(penc(x^p, pk(y^p_{nid})));$
$A^{sign}.\ att(x^p) \wedge att(sk(y^p_{nid})) \rightarrow att(sign(x^p, sk(y^p_{nid})));$
$A^{mac}.\ att(x^p) \wedge att(k^p(y^p_{nid_i}, y^p_{nid_j})) \rightarrow att(mac(x^p, k^p(y^p_{nid_i}, y^p_{nid_j})));$
$A^{add}.\ att([List^p]) \wedge att(y^p) \rightarrow att([List^p, y^p]);$
$A^{comp}.\ att(t^p_1) \wedge \cdots \wedge att(t^p_m) \rightarrow att((t^p_1; \ldots; t^p_m));$

$A^{dcom}.\ att((t^p_1; \ldots; t^p_m))$ derives additional m rules:
$att((t^p_1; \ldots; t^p_m)) \rightarrow att(t^p_1), \ldots,\ att((t^p_1; \ldots; t^p_m)) \rightarrow att(t^p_m);$
$A^{msg}.\ att((head; v_1; \ldots; [List]; \ldots; v_k))$ yields additional $k + 2$ rules
$A^{msg}_{att(head)}.\ att((head; v_1; \ldots; [List]; \ldots; v_k)) \rightarrow att(head),$
$\vdots$
$A^{msg}_{att([List])}.\ att((head; v_1; \ldots; [List]; \ldots; v_k)) \rightarrow att([List]),$
$\vdots$

$A^{msg}_{att(v_k)}$. $att((head; v_1; \ldots; [List]; \ldots; v_k)) \rightarrow att(v_k)$;

$A^{contain}$:

$A^{contain}_{List_1}$. $att([List, l^p[\ ]]) \rightarrow att([List])$;

$A^{contain}_{List_2}$. $att([l^p[\ ], List]) \rightarrow att([List])$;

$A^{replace}$:

$A^{replace}_{ID}$. $att((rreq; l^p_{src}; l^p_{dst}; t^p_{incorrect}; v_1; \ldots; [List]; \ldots; v_k)) \rightarrow att(t^p_{incorrect})$;

Let the set of all attacker's rules be $\mathcal{C}_{att}$. Rule A^{data} says that the attacker node can create arbitrary new data n^p such as fake ID identifiers, where s is a session ID to identify the session in which data was generated. Rule A^{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^{hash} says that if the attacker has some pattern x^p it can compute its hash value $h(x^p)$. Rule A^{senc} and A^{penc} 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^{sign} and A^{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 posseses, respectively. A^{add} says that the attacker can append a node ID to the end of the list $[List]$, where $List$ can be empty. Finally, rule A^{comp} is a composition rule, which says that the attacker is capable to compose a tuple of the patterns it owns.

Rule A^{dcom} 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^{msg} is similar to A^{dcom} but instead of decomposing a tuple of arbitrary patterns it deliberately considers a request and reply messages, which have a specific form, and decomposes them to smaller parts. We define reply and request as the form $(head; v_1; \ldots; [List]; \ldots; v_k)$, which is usually valid in case of source routing protocols. The part $head$ represents the head ($rrep/rreq$, l^p_{src}, l^p_{dest}, ID) of reply and request messages, respectively. The $[List]$ 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 $A^{contain}_{List_1}$ and $A^{contain}_{List_2}$ say that if the attacker has an ID list $[List, l^p[\ ]]$ and $[l^p[\ ], List]$, respectively, where $l^p[\ ]$ is new node ID that has not occured before during the verification procedure, then the attacker has the sublist $[List]$. Let the set of all $A^{contain}_{List}$-type rules be CONTAIN. We can add more rules into CONTAIN to examine the case when the new ID is inserted to every possible places in the ID list. Rule $A^{replace}_{ID}$ says that if the attacker has a request/reply in which an incorrect typed (i.e., not a node ID type) data, $t^p_{incorrect}$, is located in the place of the node ID, then the attacker has $t^p_{incorrect}$. Let the set of all $A^{replace}$-type rules be REPLACE. The difference between the rules in A^{msg} and $A^{replace}$ is that in A^{msg} the message parts are located in correct places within the request/reply. Namely, the type of the data exptected at a given place within a request or reply matches the type of the data which is residing in that place.

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

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

9.6 Derivation

Resolution: The verification is based on a guided execution of *resolution* steps. The notion of resolution [20] is well-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, we review the (well-known) notion of *substitution* and *unification*. Substitution binds some pattern to some variable, and it is typically denoted by σ. For instance, $\sigma = \{y^p_{nid} \leftarrow l^p_{src}\}$ binds constant l^p_{src} to the variable y^p, we note that σ 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 σ such that $F_1\sigma = F_2\sigma$, where $F\sigma$ represents the application of σ to F. The substitution σ that unifies two facts is called as an unifier.

As mentioned at the end of Section 9.4.3, verification procedures such as hash and MAC verification are implicitly modelled by unification. To illustrate this let us take the following example rule: $wm((l^p_{src}, [l^p_1], MAC([l^p_1], k(l^p_{src}, l^p_{dest})))) \rightarrow accept([l^p_1])$. 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, lets say that during the automatic reasoning the fact $wm((l^p_{src}, [l^p_1], MAC([l^p_1], k(l^p_1, l^p_2))))$ 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 15. *A substitution σ is a most general unifier (mgu) of a set of facts S if it unifies S, and for any unifier ω of S, there is a unifier λ such that $\omega = \sigma\ \lambda$.*

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

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 $R{\circ_F}F$ between a rule R and a fact F, where $R = F_1 \wedge \ldots \wedge F_n \rightarrow C$, and F is unifiable with C. As the result of $R{\circ_F}F$, after F and C are unified with σ, they are eliminated and the hypothesis of R, $F_1\sigma \wedge \ldots \wedge F_n\sigma$, is yielded.

A rule R of form $F_1 \wedge \ldots \wedge F_n \rightarrow C$ 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 labelled 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 $F_1\sigma$, ..., $F_n\sigma$, where σ is the most general unifier of F and C. As an example, in Figure 21 the trees in case of the rules A^{hash}, A^{mac}, A^{sign}, and $R^{rep}_{1.1}$ 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 $att(h(x^p))$: Let us consider the tree corresponds to the rule A^{hash} in Figure 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 $att(x^p)$ and then continued with proving each of the facts that are required to derive $att(x^p)$, and so on.

Definition 17. *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 ϵ be a fact that takes its value from $\{I^{ownID}_{att} \cup I^{sKey}_{att} \cup I^{pKey}_{att} \cup R^{req}_1\}$. 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 labelled with the rules in $\mathcal{R}$, and all the leaves are ϵ.*

Intuitively, the derivation tree represents a consecutive execution of resolution steps. Next, we introduce some notions that we will use when describing the verification algorithm. Whenever

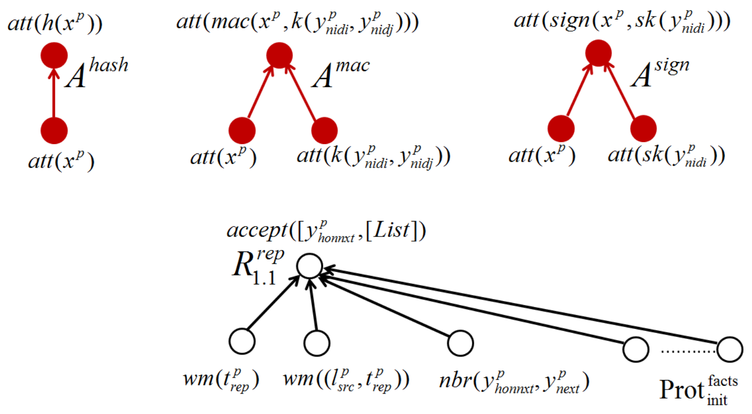

$att(h(x^p))$
A^{hash}
$att(x^p)$
$att(mac(x^p, k(y^p_{nidi}, y^p_{nidj})))$
A^{mac}
$att(x^p)$
$att(k(y^p_{nidi}, y^p_{nidj}))$
$att(sign(x^p, sk(y^p_{nidi})))$
A^{sign}
$att(x^p)$
$att(sk(y^p_{nidi}))$
$accept([y^p_{honnxt}, [List])$
$R^{rep}_{1.1}$
$wm(t^p_{rep})$
$wm((l^p_{src}, t^p_{rep}))$
$nbr(y^p_{honnxt}, y^p_{next})$
$\mathrm{Prot}^{\mathrm{facts}}_{\mathrm{init}}$

Figure 21: The trees correspond to the attacker rules A^{hash}, A^{mac}, A^{sign}, and the protocol rule $R^{rep}_{1.1}$. The derivation trees corresponding to the attacker rules are filled.

a resolution step $R \circ_F F$ is executed during the backward search, we 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, we 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, we say that F is *breadth-extended*.

9.7 The verification algorithm

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 stronger attacker computation ability can be assumed. Using backward reasoning in [21] the authors have been succeeded in verifying the occurence of *loop* in mobile ad-hoc networks assuming an arbitrary topology. The authors used graph transformation, which is suitable for modelling pure routing protocols without cryptography, however, it is not well-suited for modelling secure routing protocols. To cope with an arbitrary topology and secure routing protocols at the same time we combine the advantage of backward searching and logic based deduction.

At the beginning, an invalid route $[List_{invalid}]$ is assumed to be accepted by the source, which is expressed by the fact $accept([List_{invalid}])$. Afterwards, we follow the way of the reply which contains this invalid list in a backward manner. The possible paths of this reply is investigated by reasoning about the nodes and edges through which this reply and the corresponding request should 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. If this is the case, at this point we are aware of the information of which message should the attacker forward to be accepted later, that is, what messages should the attacker generate in order to perform a successful attack. This is then followed by examining how the attacker can generate these messages.

The attacker is able to compose a reply or request message using its computational ability and knowledge base. We note that while the computation ability of the attacker is fix, its knowledge base is continually updated during the route discovery. 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 initial knowledge? (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 initial knowledge? (iii) How the topology should be formed such that the attacker is able to intercept the required message parts?

The verification ends either when we have reached back to the state in which the source node generates the initial request, or in every searching branch the search gets stuck. In the first case, an attack scenario is found whilst in the second case no attack is found regarding the invalid route $[List_{invalid}]$. In the following, we give an overview of the algorithm in more details:

Algorithm: The input of the algorithm is the maximal length of the invalid route, n. During the verification the algorithm examine the invalid route from length 1 to n. The attack scenario is stored in the tuple of three sets $(\mathcal{T}_{prot}, \mathcal{M}_{msg}, \mathcal{A})$, where $\mathcal{T}_{prot}$ is used to store the "attack topology" in which the attack has been detected, $\mathcal{M}_{msg}$ is a set of the messages exchanged by honest nodes during the attack, and $\mathcal{A}$ stores messages sent by the attacker. At the end, in case an attack is detected the attack scenario $(\mathcal{T}_{prot}, \mathcal{M}_{msg}, \mathcal{A})$ is returned, otherwise, $(\emptyset, \emptyset, \emptyset)$ is returned.

At first, $\mathcal{T}_{top}$ includes only three nodes: The attacker node $l^p_{att_i}$, the destination node l^p_{dest}, and the source node l^p_{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, $\mathcal{T}_{top}$ is updated with them. In addition, the exchanged request and reply messages that are concerned with the discovered route are added to the set $\mathcal{M}_{msg}$, and the attacker behaviour is continually tracked by updating $\mathcal{A}$. $\mathcal{T}_{prot}$ is updated after each resolution step $R \circ_F F$, where the hypothesis of R involves at least one *nbr*-fact. In particular, the update of $\mathcal{T}_{prot}$ is composed of adding *nbr*-facts in the hypothesis *Hyp* that is resulted from $R \circ_F F$ into $\mathcal{T}_{prot}$, which is followed by removing the *nbr*-facts from *Hyp*. The update of $\mathcal{M}_{msg}$ and $\mathcal{A}$ 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 analoguos interpretation for derivation tree is as follows: Node F is depth-extended with rule R, and $\mathcal{T}_{prot}$, $\mathcal{M}_{msg}$, $\mathcal{A}$ are updated and sanitized. Finally, the *nbr*-facts among the children of F and their corresponding edges are removed from the tree.

The logic rules are classified into four subsets based on the different scenarios they specify. Set $\mathcal{S}_{hon}$ includes the request and reply rules concerning the scenarios in which the messages are exchanged between only honest nodes. The rules describing the case when the attacker node receives or overhears a message are put into $\mathcal{S}^{Recv}_{att}$. Set $\mathcal{S}^{Fw}_{att}$ 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^{Comp}_{att}.

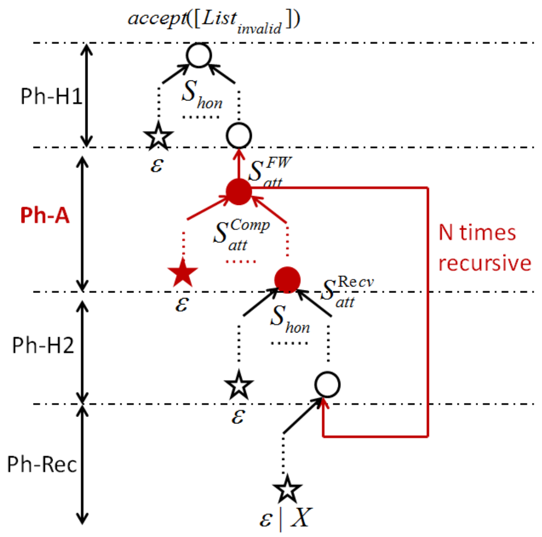


Figure 22: *The figure shows the general derivation tree occur during verification. The rulesets placed at each phase and edges represent that in that phase they are used by the algorithm to extend the tree. The notation $\epsilon \mid X$ says that the derivation can either terminate successfully or gets stuck.*

As mentioned before, at the beginning we assume that the fact *accept*$([List_{invalid}])$ has been derived. The route $[List_{invalid}]$ is formally defined by the set $\mathcal{T}_{invalid}$ that is composed of *nbr*-facts.

For example, $[l_1^p]$ is defined by set $\{nbr(l^p_{src}, l^p_1),\ nbr(l^p_1, l^p_{src}),\ nbr(l^p_1, l^p_{dest}),\ nbr(l^p_{dest}, l^p_1)\}$. During the verification procedure, whenever the topology $\mathcal{T}_{prot}$ is updated, the algorithm checks if $\mathcal{T}_{invalid} \subseteq \mathcal{T}_{prot}$ holds. If the answer is yes, we get a contradiction because the route $[List_{invalid}]$ is valid in $\mathcal{T}_{prot}$, which means that $\mathcal{T}_{prot}$ is not an attack topology. Hence, in the description of the algorithm below, by derivation of some fact we mean such a derivation during which $\mathcal{T}_{invalid} \subseteq \mathcal{T}_{prot}$ never holds whenever $\mathcal{T}_{prot}$ is updated. The rest part of the verification is concerned with searching for a derivation of the fact $accept([List_{invalid}])$. Figure 22 shows the general forms of the derivation tree that may occur during the backward searching procedure.

In *Ph-H1* the algorithm investigates how the reply, which is passed on by the attacker, propagates from the attacker to the source. Note that according to Corollary 1 the attacker must pass on replies in order to perform an attack. The reasoning about how the attacker could generate an incorrect reply *Rep* which leads to a successful attack takes place in phase *Ph-A*. The derivation of $accept([List_{invalid}])$ is successfully terminated at the end of *Ph-A* in case the attacker is able to create *Rep* based solely on its computation ability and initial knowledge. Otherwise, if some part v_i of *Rep* cannot be generated directly by the attacker, the verification is continued with phase *Ph-H2*.

In the honest phases the tree is continually extended using the rules in $\mathcal{S}_{hon}$. Within the attacker phases the tree is extended with the rules in $\mathcal{S}^{Comp}_{att}$. The edges labelled by $\mathcal{S}^{FW}_{att}$ and $\mathcal{S}^{Recv}_{att}$ in Figure 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 $\mathcal{S}^{FW}_{att}$ and $\mathcal{S}^{Recv}_{att}$, respectively. Let us call the subtree in *Ph-A* as *PhA*-tree. The root of Ph2-tree is a fact of form $att(rep)$ or $att(req)$, where both *rep* and *req* are of form $(head; v_1; \ldots; [List]; \ldots; v_k)$. In our method we attemp 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 $(head; v_1; \ldots; [List]; \ldots; v_k)$.

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

```
1.  if ($F_{wm}$ is the fact $wm(t^p_{reqinit})$) then return ($\mathcal{T}_{top}$, $\mathcal{M}_{msg}$, $\mathcal{A}$); else
    beginELSE
    if $F_{wm}$ has a request type then beginIF $\mathcal{S}$:=$\mathcal{S}^{req}_{hon} \cup \mathcal{S}^{Fw}_{att}$; goto point 2. endIF
    else if $F_{wm}$ has a reply type then beginELSE $\mathcal{S}$:=$\mathcal{S}^{rep}_{hon} \cup \mathcal{S}^{Fw}_{att}$; goto point 2. endELSE
2.    Choose a rule $R \in \mathcal{S}$ with which $F_{wm}$ can be extended & not applied to $F_{wm}$ before;
3.    if ($\nexists\ R \in \mathcal{S}$ that has not been chosen before for extending $F_{wm}$) then
4.      if ($F_{wm}$ has a wm-fact typed father, $F^{wm}_{father}$) then goto 1. with $F_{wm} := F^{wm}_{father}$;
5.      else if ($F_{wm}$ has a att-fact typed father, $F^{att}_{father}$) then step into Ph-A with $F_{att} := F^{att}_{father}$;
6.      else return ($\emptyset$,$\emptyset$,$\emptyset$);
      else
      beginELSE
7.      Let denote the set of nbr-facts among the children of $F_{wm}$ by nbrfacts($F_{wm}$);
8.      Update $\mathcal{T}_{top}$ with nbrfacts($F_{wm}$);
9.      if ($\mathcal{T}_{invalid} \subseteq \mathcal{T}_{top}$) then remove nbrfacts($F_{wm}$) from $\mathcal{T}_{top}$; goto 1.; else
        beginELSE
10.       Let denote the set of wm-facts among the children of $F_{wm}$ by wmfacts($F_{wm}$);
11.       Let denote the set of att-facts among the children of $F_{wm}$ by attfacts($F_{wm}$);
12.       Remove nbr-facts in nbrfacts($F_{wm}$) from the tree;
13.       update $\mathcal{M}_{msg}$, $\mathcal{A}$ with the facts from wmfacts($F_{wm}$) and attfacts($F_{wm}$), respectively;
14.       goto 1. with $F_{wm}$ := a fact in wmfacts($F_{wm}$),
          or step into phase Ph-A with a att-fact $F_{att}$ in attfacts($F_{wm}$);
        endELSE
      endELSE
    endELSE
```

The explanation of the pseudo-code for phase *Ph-H* is as follows (Table 1): This phase is defined by the function $PhH(F_{wm})$, in which we assume that the derivation procedure has reached some *wm*-fact or the fact $accept([List_{invalid}])$, denoted by F_{wm}, and we continue with searching for a possible derivation of F_{wm}. The searching procudure is basically a Depth-First search.

At the beginning, if F_{wm} is the fact $wm(t^p_{reqinit})$, where the pattern $t^p_{reqinit}$ represents the initial request sent by the source, then the deduction procedure terminated successfully, and the attack scenario ($\mathcal{T}_{top}$, $\mathcal{M}_{msg}$, $\mathcal{A}$) 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 $R^{header}_{req} \circ_{F_{wm}} F_{wm}$, and $R^{header}_{rep} \circ_{F_{wm}} F_{wm}$. 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 ϵ 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 an 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 an 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. $F_{wm} = accept[List_{invalid}]$, then not any derivation for F_{wm} can be found. This means that no attack scenario is detected, and (∅,∅,∅) is returned (point 5).

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

Points 9-13 consider the case when $\mathcal{T}_{invalid} \nsubseteq \mathcal{T}_{top}$, and we put the *wm*-facts and *att*-facts in the sets *wmfacts*(F_{wm}) and *attfacts*(F_{wm}) into $\mathcal{M}_{msg}$ and $\mathcal{A}$, 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 2: Pseudo-code of the attacker phase, function *PhA*(F^{root}_{att}).

```
MAIN PART

a1. Decompose $F^{root}_{att}$ with $A^{comp}$; Put the resulted facts into $W$;
a2. if ($\nexists\ F_{att} \in W$ which has not been eximaned yet) then
    beginIF
a3.   if ($\forall$ leaf in the tree is $\epsilon$) then return ($\mathcal{T}_{top}$, $\mathcal{M}_{msg}$, $\mathcal{A}$);
      else step into the honest phase $PhH(F_{wm})$, where $F_{wm}$ is a wm-fact and the father of $F^{root}_{att}$;
    endIF else
    beginELSE
a4.    Choose $F_{att} = att(t^{p}_{att})$: $t^{p}_{att}$ has the highest priority and weight in $W$;
     /* If $t^{p}_{att}$ is a constant, key, ID OR a keyless function (e.g., hash) */
a5.      if ($F_{att}$ has no child OR $\forall$ child of $F_{att}$: not key-type) then
a6.        $FNotKeyedFunc(F_{att})$;
         endIF else
         beginELSE
     /* If $t^{p}_{att}$ is a keyed function (e.g., digital signature, MAC), and the attacker knows the key */
a7.        if ($\exists$ child $F'_{att}$ of $F_{att}$, $F'_{att} \in I^{sKey}_{att} \cup I^{pKey}_{att}$) then
           beginIF
a8.          $FAttKeyedFunc(F_{att})$;
           endIF else
           beginELSE
     /* If $t^{p}_{att}$ is a keyed function, but the attacker does not know the key */
     /* Examine how the attacker can obtain $t^{p}_{att}$, that is, which REQ/REP contains it */
a9.          Extend $F_{att}$ with $A^{msg}_{F_{att}}$ results in some $F^{msg}_{att} = att(t^{msg}_{att})$;
     /* Examine how the attacker can obtain the request/reply $t^{msg}_{att}$ that contains $t^{p}_{att}$ */
a10.         if ($F^{msg}_{att}$ has NOT been examined before) then
             beginIF
a11.           $REPREQcorrectplace(F_{att})$;
             endIF else
             beginELSE
a12.           $REPREQincorrectplace(F_{att})$;
             endELSE
           endELSE
         endELSE
    endELSE
```

MAIN part: F^{root}_{att} is an *att*-fact $att(t^{root}_{att})$, where the pattern t^{root}_{att} is a reply or request message. The index *root* of F^{root}_{att} refers to the root of the current Ph-A sub-tree. First, F^{root}_{att} is decomposed with the rule A^{Comp}, and the resulted *att*-facts are put into W (point a1). An attack scenario is returned in case no unexamined facts are left (i.e., $W = \emptyset$), and all the deduction branches terminate successfully, otherwise, we return to the father of F_{att}^{root}, F_{wm}, and search for another deduction path (points a2-a3). If there still are unexamined *att*-facts in W, the algorithm chooses the fact F_{att}, $F_{att} = att(t_{att}^{p})$, such that the pattern t_{att}^{p} has the highest priority and the highest weight, which has not been examined before during the verification (point a4).

Then, the deduction procedure is continued based on the type of t_{att}^{p}. The function *FNotKeyed-Func*(F_{att}) is responsible for case when t_{att}^{p} is a constant, a node ID, a session ID, or a keyless crypto function (point a6). Function *FAttNotKeyedFunc*(F_{att}) handles the case when t_{att}^{p} is a keyed crypto function, and the key is owned by the attacker (point a8). If t_{att}^{p} is a keyed 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_{att}^{p} (points a9-a12). Function *REPREQcorrectplace*(F_{att}) analyzes the scenario when t_{att}^{p} is in a correct place within a request or reply message, while *REPREQincorrectplace*(F_{att}) considers the case when t_{att}^{p} is located in an incorrect place (for example, a MAC is put into the place of a session ID).

Func. ***FNotKeyedFunc***(F_{att})

```
6.1.    if ( #childs($F_{att}$) > 1) then
        beginIF
6.2.      Put all the children of $F_{att}$ into $W$; and perform deduplications.
6.3.      Delete $F_{att}$ from $W$; Goto point 2. in MAIN;
        endIF else
6.4.    if ($F_{att}$ has no child OR #childs($F_{att}$) = 1) then
        beginIF
6.5.      if ($F_{att}$ OR the child of $F_{att}$ $\in$ Knowledge of the attacker) then
          beginIF
6.6.        Delete $F_{att}$ from $W$; Goto point 2. in MAIN;
6.7.      endIF else Goto point 9. in MAIN;
        endIF
```

Func. ***FAttKeyedFunc***(F_{att})

```
      /* Let $att(Data) \wedge att(key) \rightarrow att(t^p_{att})$, where $F_{att} = att(t^p_{att})$ */
8.1       Decomposing $att(Data)$ with $A^{comp}$ yields some factset $W'$;
8.2       Put $W'$ into $W$ and then eliminate fact duplication in $W$;
8.3       Delete $F_{att}$ from $W$; Goto point 2. in MAIN;
```

Func. ***REPREQcorrectplace***(F_{att})

```
11.1   Extend $F^{msg}_{att}$ with some rule $R$, $R \in \mathcal{S}^{Recv}_{att}$, yielding a $wm$-fact $F_{wm}$, and step into $PhH(F_{wm})$;
11.2   if (In $PhH(F_{wm})$ the search does not get stuck, i.e., every leaf is $\epsilon$ & $\mathcal{T}_{invalid} \nsubseteq \mathcal{T}_{top}$) then
       beginIF
11.3     Decompose $F^{msg}_{att}$ which yields some $W'$;
11.4     $W = W - \{W' \cup F_{att}\}$; Goto point 2. in MAIN;
       endIF
       else if (we gets stuck in $PhH(F_{wm})$, and step back into $PhA(F^{msg}_{att})$ according to point 5. in $PhH(F_{wm})$)
       beginELSE
11.5      Decompose $F^{msg}_{att}$ which yields some $W'$;
11.6      while ($\exists$ rule $A^{contain}_{List} \in$ CONTAIN that has not been applied) do
          beginWHILE
11.7        Extend $att([List]) \in W'$ with $A^{contain}_{List}$ yielding $att([List, l^p[\,]])$;
11.8        Extend $att([List, l^p[\,]])$ with $A^{msg}_{List}$ yields some $att(t^{broader}_{att})$;
11.9        if ($att(t^{broader}_{att})$ has NOT been examined before) then
            beginIF
11.10         Decompose $att(t^{broader}_{att})$ which yields some $W''$;
11.11         if ($F_{att} \in W''$) then
              beginIF
11.12           Extend $att(t^{broader}_{att})$ with some $R$, $R \in \mathcal{S}^{Recv}_{att}$, yielding a $wm$-fact $F_{wm}$, and step into $PhH(F_{wm})$;
11.13           if (In $PhH$ the search does not get stuck, i.e., every leaf is $\epsilon$ & $\mathcal{T}_{invalid} \nsubseteq \mathcal{T}_{top}$) then
                beginIF
11.14             $W = W - \{W'' \cup F_{att}\}$;
11.15             Terminate Loop & Goto point 2. in MAIN;
                endIF
                else if (the search steps back into $PhA(att(t^{broader}_{att}))$ according to point 5. in $PhH(F_{wm})$)
                  then goto point 11.6.
              endIF
            endIF
          endWHILE
       endELSE
11.16  Goto point 12. in MAIN;
```

Func. ***REPREQincorrectplace***(F_{att})

```
12.1    while ($\exists$ $A^{replace}_{F_{att}} \in$ REPLACE that has not been applied) do
        beginWHILE
12.2      Extend $F_{att}$ with $A^{replace}_{F_{att}}$ results in some $att(t^{replace}_{att})$;
12.3      if ($att(t^{replace}_{att})$ has NOT been examined before) then
12.4        Extend $att(t^{replace}_{att})$ with some rule $R$, $R \in \mathcal{S}^{Recv}_{att}$, yielding a $wm$-fact $F_{wm}$, and step into $PhH(F_{wm})$;
12.5        if (In $PhH$ the search does not get stuck, i.e., every leaf is $\epsilon$ & $\mathcal{T}_{invalid} \nsubseteq \mathcal{T}_{top}$) then
            beginIF
12.6          Decompose $att(t^{replace}_{att})$ which yields some $W'$;
12.7          $W = W - \{W' \cup F_{att}\}$; Terminate Loop & Goto point 2. in MAIN;
            endIF
            else if (the search steps back into $PhA(att(t^{replace}_{att}))$ according to point 5. in $PhH(F_{wm})$)
              then goto point 12.1.
        endWHILE
12.8    step into the honest phase $PhH(F_{wm})$, where $F_{wm}$ is a $wm$-fact and the father of $F^{root}_{att}$;
```

In function $FNotKeyedFunc(F_{att})$, if F_{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_{att} is removed from W (to avoid deduction loop), and we go on with analyzing the following *att*-fact in W (points 6.2-3.). If F_{att} does not have any child (i.e., t^p_{att} is a constant, a node ID, a session ID), or it has exactly one child (i.e., t^p_{att} is a keyless function such as a one-way hash function) (point 6.4.). In this case, we check whether F_{att} or its only child is an element of the knowledge base of the attacker (i.e., whether F_{att} belongs to ϵ) (point 6.5.). If yes, then this deduction branch terminates successfully (point 6.6.), otherwise, we reason about how the attacker can obtain the request or reply that contains t^p_{att} (point 6.7.).

In function *FAttKeyedFunc*(F_{att}), where t^p_{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 8.1.), then we update W with the resulted *att*-facts (point 8.2.). Finally, F_{att} are removed from W (point 8.3.).

Function *REPREQcorrectplace*(F_{att}) examines how the attacker node can intercept or receive the request/reply message that contains t^p_{att}, in a correct place within a request or reply message. Let denote the request and reply that contains t^p_{att} be t^{msg}_{att}. At the beginning, we search for the derivation of F^{msg}_{att}, $F^{msg}_{att} = att(t^{msg}_{att})$, in the honest phase Ph-3 (point 11.1). If F^{msg}_{att} can be successfully derived (point 11.2.), then we remove all the parts of the message t^{msg}_{att} from W, and start to examine the next fact in W (points 11.3-4.). If the derivation of F^{msg}_{att} got stuck (either because some deduction branch terminates with a fact that is not in ϵ, or $\mathcal{T}_{invalid} \subseteq \mathcal{T}_{top}$), then as the following step, we reason about whether a broader request/reply message (that contains a longer ID list) can contain t^p_{att}. The rule $A^{contain}_{List}$, $A^{contain}_{List} \in$ CONTAIN, specifies the action that inserts a (new) honest node ID, $l^p[\,]$, into the list *List*. In our case, for verifying SRP, Ariadne and EndairA, it is sufficient to define only one $A^{contain}_{List}$ rule that insert $l^p[\,]$ into the end of *List* (point 11.7.). In general, the set CONTAIN may consist of more rules that insert more node IDs into different places in *List*. In point 11.8, by extending $att([List, l^p[\,]])$ with the rule A^{msg}_{List} we get $att(p^{broader}_{att})$ as result, where $t^{broader}_{att}$ is the "broader" request/reply message that contains the ID list $[List, l^p[\,]]$. In case $t^{broader}_{att}$ has not been examined before, we check if it contains t^p_{att} (points 9-11). If yes, we reason about how the attacker can obtain the message $t^{broader}_{att}$ (points 12-15).

If the derivation got stuck in case of all the defined rules in CONTAIN, then we proceed to *REPREQincorrectplace*(F_{att}), where we examine the possibility that t^p_{att} (recall that $F_{att} = att(t^p_{att})$) is located in an incorrect place (for example, putting a MAC or signature into the place of session ID). The rule $A^{replace}_{F_{att}}$ in the set REPLACE, defined for F_{att}, specifies the insertation of t^p_{att} to incorrect places. The pattern $t^{replace}_{att}$ represents the request/reply message, in which the message element t^p_{att} is located in an incorrect place. Again, for our purpose, it is sufficient to define one such rule, which inserts data to the place of the session ID.

9.8 Termination

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

We assume on-demand source routing protocols, where the ID list is placed in the request and reply messages. We 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 session ID once, which is valid in the most well known on-demand source routing protocols. In our verification procedure, we ensure this with the following two assumptions:

1. We examine only such invalid route $List_{invalid}$ in $accept([List_{invalid}])$, 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 against invalid route when we design a routing protocol. Note that we can extend our deduction algorithm for detecting loop during the protocol run, but this falls outside the focus of the dissertation, where we 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. We 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 valid to the messages sent by the attackers). This assumption is valid to all the well-known 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 does 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 in the request is broadcast unchanged by the honest nodes, then in the protocol rule

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

   we cannot determine which l^p_i is sender. Hence to make the deduction algorithm usable, we have to add the ID of the sender into the *wm*-facts, namely, $wm((l^p_i,\ y^p_{req}))$. Note that the added ID is not part of the request message, and only serves the automated deduction purpose: $wm(l^p_i,\ y^p_{req})) \wedge\ nbr(l^p_i,\ l^p_j) \rightarrow wm(l^p_j,\ y^p_{req}))$.

Lemma 2. *Besides the assumptions we provided above, in the honest phase PhH(F_{wm}), the deduction will not step into an infinite loop.*

Proof. First of all, point 3 of $PhH(F_{wm})$ prevents the usage of the same rule R for the a given F_{wm} infinite amount of time, 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, we will show that during the tree extending process (in a 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 deduction (tree) branch will not contain the same F_{wm} again, infinitely. Within *Ph-H* we further distinguish the request and reply phases in which the request and reply messages are exchanged between honest nodes, respectively.

- During a request phase, the message exchanges among honest nodes are simulated by the protocol rules R^{req}_3, R^{req}_2 and R^{req}_1. After each (backward) resolution step, $R^{req}_3 \circ_{F_{wm}} F_{wm}$, where $F_{wm} = wm(t^p_{reqj})$ and $R^{req}_3 = wm(y^p_{reqi}) \wedge\ nbr(l^p_i,\ l^p_j) \rightarrow wm(y^p_{reqj})$, we get $wm(t^p_{reqi}) \wedge\ nbr(l^p_i,\ l^p_j)$ as result. This means that in order to make l^p_j able to send the request t^p_{reqj}, its neighbor node l^p_i, should have sent the request t^p_{reqi}. Based on the assumption that $List_{invalid}$ contains finite number of different node IDs, and the protocol rules R^{req}_3, R^{req}_2 and R^{req}_1 specify message exchanges between different nodes, it follows that the deduction procedure will terminate within a finite number of resolution steps. The resolution steps will be performed constantly using R^{req}_3 until we reach to the point when the initial request has been sent by the source node, where the rules R^{req}_2 and R^{req}_1 are used.

- The situation is similar in case of the reply phase. Again, let the reply $t^p_{rep'}$ that has been sent by node y^p_{i-1} to y^p_{prev} includes the ID list $[List, y^p_{prev}, y^p_{i-1}, y^p_{next}, List]$. The algorithm searches for the rules in $\mathcal{S}_{hon}$ to extend the tree at $wm((y^p_{prev}, t^p_{rep'}))$. After extending $wm((y^p_{prev}, t^p_{rep'}))$ the fact $wm((l^p_{i-1}, t^p_{rep}))$ is yielded. We recall that the ID at the beginning of the message represents the addressee of the reply. Due to the value of the addressee 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. The resolution steps will be performed consecutively using $R^{rep}_{2.12}$ until we reach to the point when the destination node sent back a reply after it received a request, which is modelled by the resolution with the rule R^{req}_{4}.

To summarize, in both the request and reply phases, after performing each resolution step (i.e., tree extending step) basically 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 a finite number of resolution steps.

$\square$

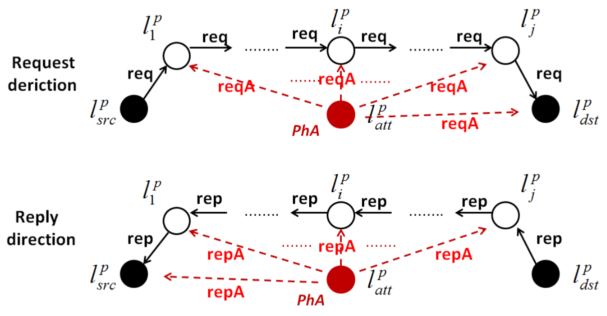


Figure 23: The possibilities to get into an attacker phase *PhA* during the request and reply directions.

We continue with showing that the attacker phase $PhA(F^{root}_{att})$ is infinite loop-free as well.

Lemma 3. *During searching for a derivation of accept*$([List_{invalid}])$ *the algorithm does not get into an infinite deduction loop in the attacker phase* $PhA(F^{root}_{att})$.

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^{comp} and A^{dcomp} are performed iteratively in turn. However, in our method we prevent this by performing decomposition only, and instead of using composition rule to set up the required message we introduced the set of rules A^{msg}_{i}, 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 $A^{contain}_{List}$ may be performed infinite time. However, this is not the case because each application of $A^{contain}_{List}$ introduced a new node in the network, which contains only finite number of nodes. Formally, $A^{contain}_{List}$ is allowed only to applied up to the number of nodes in the network.

In $PhA(F^{root}_{att})$ we have to examine and search for the derivation of the *att*-facts placed in W. The *att*-facts in W are the parts of the request and reply, and its number is finite because the request and reply messages contain finite data 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 (which is finite).

In addition, we prevent deduction loop by also keeping track of the *att*-facts that are already examined before during the current deduction procedure (point 1 of $PhA(F^{root}_{att})$), and whenever we get into the point where we need to derive the same *att*-fact again, we stop continuing this deduction branch. In point 10 of $PhA(F^{root}_{att})$, and points 11.9 and 12.3, we also keep track of the request/reply messages that we have examined before. Since the number of the possible requests/replies t^{msg}_{att}, $t^{broader}_{att}$ and $t^{replace}_{att}$, and the *att*-facts in W are finite, the total number of resolutions performed in $PhA(F^{root}_{att})$ will be finite as well. $\square$

Finally, we show that during the whole deduction procedure, the occurrences of the honest and the attacker phases are finite. Namely, there will not be an infinite loop between *PhA* and *PhH*. In the request phase, whenever we search for a derivation of a given *wm*-fact F_{wm}, we can step into an attacker phase by performing a resolution with the rules R^{req}_{att5}, R^{req}_{att6}, and R^{req}_{att7}, yielding an *att*-fact F_{att}^{root} (where $F_{att}^{root} = att(t_{req}^{p})$, for some request message t_{req}^{p}). According to the deduction steps defined in *PhA*, we have to search for the derivation of every message element in t_{req}^{p}, which is finite. In the worst case, we step into phase *PhA* after points *11.6-11.7* of function *REQREPcorrectplace*, when we search for the derivation of the fact $att(t_{att}^{broader})$ within the phase *PhA*. Request message $t_{att}^{broader}$ contains the broader list than $List^{invalid}$, which we get by inserting new honest node IDs in it. Let the number of the node IDs in the ID list of $t_{att}^{broader}$ be j. In the honest phase (request direction), during get back from the destination to the source, we step into $PhA(F_{att}^{root})$ at most j times (shown in the Figure 23). Each time, after getting into $PhA(F_{att}^{root})$ we can step into the honest phase $PhH(F_{wm})$ again, 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 1-2 of $PhA(F_{att}^{root})$, $F_{att}^{root} = att(head^{req};\ v_1;\ \ldots;\ [List];\ldots;\ v_k)$, we search for the derivation of each element $att(head^{req})$, $att(v_1)$, $\ldots$, $att([List])$, $\ldots$, $att(v_k)$ 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 $[List]$ are finite). Hence, after some rounds, when we run out of the message elements that have not been examined before, the attacker phase will always get stuck at point 2., and the deduction procedure returns to phase $PhH(F_{wm})$ (point 3 of *PhA*), where we will step back to the source after at most j resolution steps using the rules R_1^{req}, R_2^{req}, R_3^{req}. The situation is similar in case of the reply direction.

9.9 Correctness and completeness

Correctness: In this subsection, we discuss about the correctness and the completeness which the verification method provides. By correctness we mean whenever an attack is detected and returned it is really an attack. This property is stated in Lemma 4.

Lemma 4. *Whenever an attack scenario ($\mathcal{T}_{prot}$, $\mathcal{M}_{msg}$, $\mathcal{A}$) is returned for $[List_{invalid}]$, (i.) the fact accept($[List_{invalid}]$) has been derived and (ii.) $\mathcal{T}_{invalid} \nsubseteq \mathcal{T}_{prot}$.*

Proof. According to the algorithm, the attack scenario is returned only when every leaf in the derivation tree of *accept*($[List_{invalid}]$) is ϵ, and when $\mathcal{T}_{invalid} \nsubseteq \mathcal{T}_{prot}$ holds whenever $\mathcal{T}_{prot}$ is updated with new edges (i.e., *nbr*-facts). Hence, in general case, if for □

Completeness: In the general case, when an arbitrary source routing protocol is assumed, our verification method is *not complete*. This means that when the verification tool finished and no attack scenario is returned, there can be still an attack. This is due to the following assumptions:

- Although our method does not assume any specific topology. The length of the invalid route, $List_{invalid}$, 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 $List_{invalid}$ of length k, then it means that the attackers cannot achieve that the source accept an invalid route of length k.

- We restrict the number of applying the same rule $A_{List}^{contain}$ in CONTAIN. Intuitively, we limit the number of *new* node IDs $l^p[\ ]$ (differ from the IDs in $List_{invalid}$) the attacker can insert into the ID list in request and reply messages. This is because the number of new nodes can be infinite. Nevertheless, we found that in case of the well-known routing protocols an attack can be found in the network after only one-time application of $A_{List}^{contain}$.

Otherwise, the deduction algorithm is exhaustive: The set of protocol rules taking into account every possible scenario. The set of rules also cover all the possible scenarios of the message exchanges between the honest and the attacker nodes. Hence, the request and reply direction can be simulated by the series of resolutions defined in the honest phase $PhH(F_{wm})$. In the attacker phase, *PhA*, the attacker can only perform an attack if it can obtain all the smaller parts of the request and reply messages. This is ensured in the first two points of *PhA*. To examine how can the attacker could obtain each part, we examine that (i) can the attacker compute it 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 reply. Finally, the attacker tries to insert node IDs into the ID list. 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.

9.10 Complexity

I assume that the length of $[List_{invalid}]$ in the fact $accept([List_{invalid}])$ 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 $PhH(F_{wm})$, checking the type of F_{wm} takes two resolution steps in the worst case (point 1). One resolution is for checking if F_{wm} is a request, and an another if it is a reply. Then, in point 2 F_{wm} is extended with one of the rules $R^{req}_{2.3}$, $R^{req}_{2.1}$ and R^{req}_{1}, which takes three resolutions. This requires at most three resolution steps. Moreover, in points 7-8, nbr-facts are resulted after each resolution step. Eliminating one nbr-fact takes one resolution step $nbr(l^p_i, l^p_j) \circ_{nbr} nbr(y^p_i, y^p_j)$. Let the number of nbr-facts in rules $R^{req}_{2.1}$ and $R^{req}_{2.3}$ be $nbr^{req}_{2.1}$ and $nbr^{req}_{2.3}$, which yield $max(nbr^{req}_{2.1}, nbr^{req}_{2.3})$ 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 $R^{req}_{2.3}$, until we reach the source node, where the two rules $R^{req}_{2.1}$ and R^{req}_{1} are applied, in this order (shown in Figure 24). In the worst case, the number of resolutions is equal to the number of ID list in the request message $t^{broader}_{att}$ in the fact $att(l^p_{att}, t^{broader}_{att})$. Let the number of IDs in this ID list be r. Hence, in the request phase, $(2 + 3\times\ max(nbr^{req}_{2.1}, nbr^{req}_{2.3})) \times max(r, k)$ resolution steps are required.

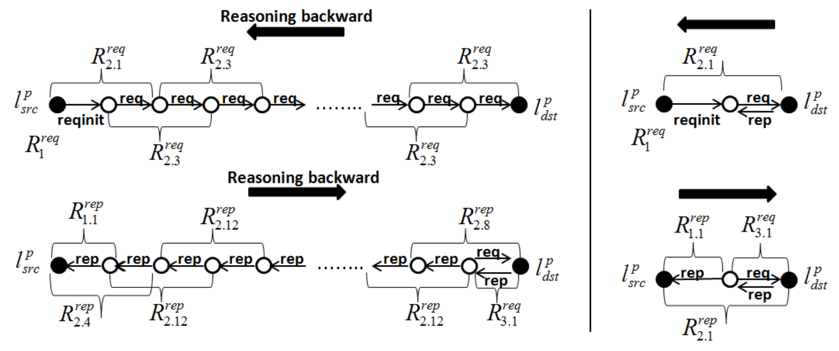


Figure 24: *The backward reasoning is based on consecutive resolution steps. After performing each resolution step with a given rule, we step back from one node to its neighbor node. On the left side, we consider the case when there is more than one intermediate node, while on the right side, the case of one honest intermediate node is illustrated.*

- *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 $\mathcal{S}^{rep}_{hon}$, $\mathcal{S}^{rep}_{hon} = \{R^{rep}_{1.1}, R^{rep}_{2.1}, R^{rep}_{2.4}, R^{rep}_{2.8}, R^{rep}_{2.12}, R^{req}_{3.1}\}$, which costs 6 resolution steps. In the reply phase, we can get back from the source to the destination by applying $R^{rep}_{1.1}$ first, followed by using the rule $R^{rep}_{2.4}$. Then, the resolution steps between the resulted fact from the previous resolutions and $R^{rep}_{2.12}$, are constantly performed. Finally, the rules $R^{rep}_{2.8}$ and $R^{req}_{3.1}$ are used to get into the request phase. In case of one intermediate node, rule $R^{rep}_{2.1}$ is applied after using $R^{rep}_{1.1}$ and before $R^{req}_{3.1}$. Therefore, in the reply phase, at most $(2 + 6 \times max(nbr^{rep}_{1.1}, nbr^{rep}_{2.1}, nbr^{rep}_{2.4}, nbr^{rep}_{2.8}, nbr^{rep}_{2.12}, nbr^{req}_{3.1})) \times max(r, k)$ resolution steps are required.

- *The complexity of the attacker phase, PhA*(F_{att}^{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 *att*-fact f_{att} in W, we search for the rule in the set of the attacker's computation ability, $\mathcal{C}_{att}$, which can be resolvable with f_{att}. This takes $|\mathcal{C}_{att}|$ number of resolution steps, where $|\mathcal{C}_{att}|$ denotes the size of $\mathcal{C}_{att}$.

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

  In point *a7* of *PhA*(F_{att}^{root}), we have to examine if the key is in the union set $I_{att}^{sKey} \cup I_{att}^{pKey}$, which requires $|I_{att}^{sKey}| + |I_{att}^{pKey}|$ resolution steps.

  In *FAttKeyedFunc*(F_{att}), the attacker rule A^{comp} is used to extend *att*(l_{att}^{p}, *Data*), which takes one resolution step. Deleting F_{att} in point *a8.3* also takes one resolution step.

  Point *a9* of *PhA*(F_{att}^{root}) costs one resolution step. Let the set that stores the already examined *att*-facts for t_{att}^{msg}, $t_{att}^{broader}$ and $t_{att}^{replace}$ messages be W_{msg}, $W_{broader}$, and $W_{replace}$, respectively. In point *a10*, we have to examine if F_{att}^{msg} has been examined before, which requires $|W_{msg}|$ resolutions.

  In *REQREPcorrectplace*, points *a11.1* and *a11.5* take $|\mathcal{S}_{att}^{Recv}|$ 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 $|W_{broader}|$ and $|W''|$ resolution steps, respectively. Finally, point *a11.12* costs $|\mathcal{S}_{att}^{Recv}|$ steps. In total, the complexity of the function is:

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

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

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

  To summarize, the worst-case complexity of phase *PhA*(F_{att}^{root}) is:

  $Complex(PhA) =$

  $1 + |W| \times (\ |W_{F_{att}}| + |\mathcal{C}_{att}| + MAX(Complex(FNotKeyedFunc), Complex(FAttKeyedFunc), |I_{att}^{sKey}| + |I_{att}^{pKey}| + MAX(Complex(REQREPcorrectplace), Complex(REQREPincorrectplace))\ )$.

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

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

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

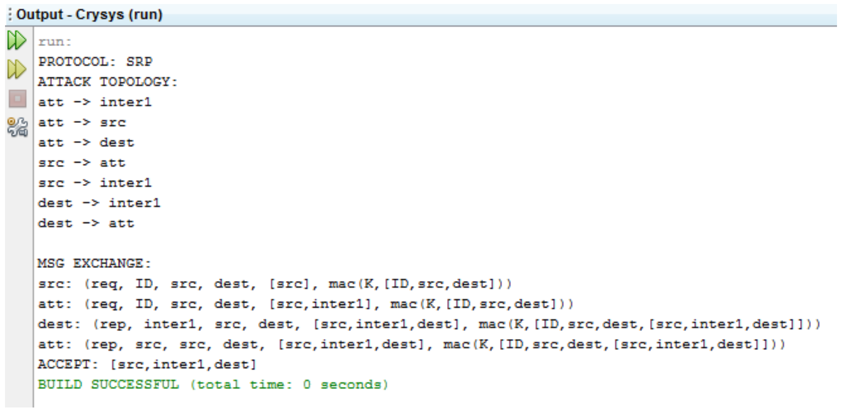
The complexity of the proposed algorithm in practise: Although in the worst case, the complexity of *sr*-verif is the exponential function of the length of the ID list $List_{invalid}$, in practise, it is very effective in case of well-known on-demand source routing protocols. In case of well-known routing protocols, an attack scenario can be found with [$List_{invalid}$] containing not more than three node IDs. An attack scenario against the DSR protocol is found when [$List_{invalid}$] = [l_1^p] is examined; an attack scenario against the SRP protocol is found at the point when the verification tool is examining the case in which [$List_{invalid}$] = [l_1^p, l_2^p], and an attack scenario against the Ariadne protocol is detected in case [$List_{invalid}$] = [$l_1^p, l_2^p, l_{att_1}^p$]. Finally, the analyzed covert channel attack against the endairA protocol (discussed in Section 8.4) can be detected based on the invalid list $[l^p_{att1}, l^p_{att2}]$, where l^p_{att1} and l^p_{att2} are the IDs of two different attacker nodes.

In practice, the complexity of the attacker phases is not large because the message elements in the *att*-fact F^{root}_{att} corresponding to a latter $PhA(F^{root}_{att})$ phase is the subset of the message elements in the upper level attacker phases and the first $PhA(F^{root}_{att})$ phase. More specifically, the *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 [5], 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 [5]. In [5] 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.

9.11 Implementation

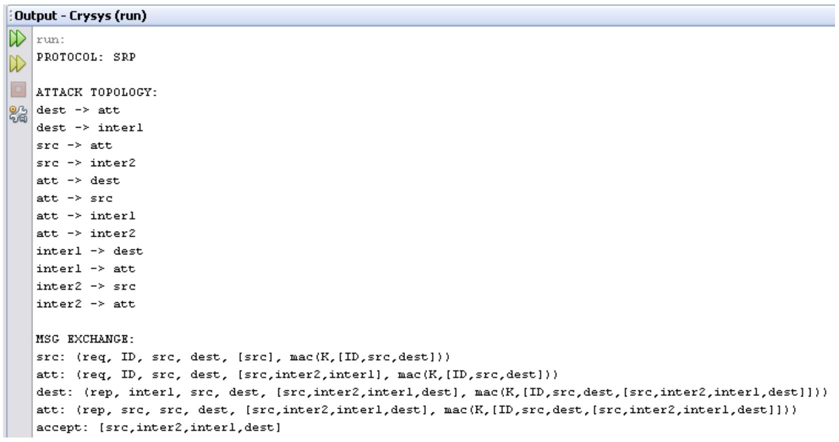
We have developed the first version of the software tool based on the theoretical foundations described in this paper. The tool is implemented in the JAVA programming language, and was succeeded in finding attack scenario in case of the SRP and Ariadne protocols. For all these examples, the total time for finding derivation trees is less than 2 seconds on a Pentium 2.8 GHz. The screenshot of the program running can be found in Figure 25 and Figure 26. The first figure is concerned with the case when uni-directional edges, whilst the latter case considers bidirectional edges. The TOPOLOGY part includes the attack topology $\mathcal{T}_{top}$, and MSG part is composed of the messages sent by honest and attacker nodes.



```
run:
PROTOCOL: SRP
ATTACK TOPOLOGY:
att -> inter1
att -> src
att -> dest
src -> att
src -> inter1
dest -> inter1
dest -> att

MSG EXCHANGE:
src: (req, ID, src, dest, [src], mac(K,[ID,src,dest]))
att: (req, ID, src, dest, [src,inter1], mac(K,[ID,src,dest]))
dest: (rep, inter1, src, dest, [src,inter1,dest], mac(K,[ID,src,dest,[src,inter1,dest]]))
att: (rep, src, src, dest, [src,inter1,dest], mac(K,[ID,src,dest,[src,inter1,dest]]))
ACCEPT: [src,inter1,dest]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 25: An attack scenario detected in case of uni-directional edges.



```
run:
PROTOCOL: SRP

ATTACK TOPOLOGY:
dest -> att
dest -> inter1
src -> att
src -> inter2
att -> dest
att -> src
att -> inter1
att -> inter2
inter1 -> dest
inter1 -> att
inter2 -> src
inter2 -> att

MSG EXCHANGE:
src: (req, ID, src, dest, [src], mac(K,[ID,src,dest]))
att: (req, ID, src, dest, [src,inter2,inter1], mac(K,[ID,src,dest]))
dest: (rep, inter1, src, dest, [src,inter2,inter1,dest], mac(K,[ID,src,dest,[src,inter2,inter1,dest]]))
att: (rep, src, src, dest, [src,inter2,inter1,dest], mac(K,[ID,src,dest,[src,inter2,inter1,dest]]))
accept: [src,inter2,inter1,dest]
```

Figure 26: An attack scenario detected in case of bi-directional edges.

10 Conclusion

We 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, we proposed a fully automatic verification method for secured ad-hoc network routing protocols, which is based on logic and a backward reachability approach. Our method has a clear syntax and semantics for modelling secure routing protocols, in addition, it handles arbitrary network topologies and considers a strong attacker model. Finally, our 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.

We have developed the first version of the software tool based on the theoretical foundations described in this paper. The tool is implemented in the JAVA programming language, and was successful in finding attack scenarios in well-known routing protocols.

Acknowledgment

The work described in this paper has been supported by the grant TAMOP - 4.2.2.B-10/12010-0009. at the Budapest University of Technology and Economics. Special thanks to my advisor, Dr. Levente Buttyán, for his encouragement, guidance, and supports.

References

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the Spi calculus. Technical Report SRC RR 149, Digital Equipment Corporation, Systems Research Center, January 1998.

[2] G. Acs, L. Buttyan, and I. Vajda. Provable security of on-demand distance vector routing in wireless ad hoc networks. In *In In Proceedings of the Second European Workshop on Security and Privacy in Ad Hoc and Sensor Networks (ESAS 2005*, pages 113–127, 2005.

[3] G. Acs, L. Buttyan, and I. Vajda. Provably secure on-demand source routing in mobile ad hoc networks. In *IEEE Transactions on Mobile Computing*, volume 5, 2006.

[4] G. Acs, L. Buttyan, and I. Vajda. The security proof of a link-state routing protocol for wireless sensor networks. In *IEEE Workshop on Wireless and Sensor Networks Security*, 2007.

[5] T. R. Andel and A. Yasinsac. Automated evaluation of secure route discovery in manet protocols. In *SPIN '08: Proceedings of the 15th international workshop on Model Checking Software*, pages 26–41, 2008.

[6] B. Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.

[7] A. Bloch, M. Frederiksen, and B. Haagensen.

[8] L. Buttyán and T. V. Thong. Formal verification of secure ad-hoc network routing protocols using deductive model-checking. In *Proceedings of the IFIP Wireless and Mobile Networking Conference (WMNC)*, pages 1–6, Budapest, Hungary, October 18-20 2010. IFIP.

[9] L. Buttyán and I. Vajda. Towards provable security for ad hoc routing protocols. In *SASN '04: Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, pages 94–105, 2004.

[10] C. Fournet and M. Abadi. Mobile values, new names, and secure communication. In *In Proceedings of the 28th ACM Symposium on Principles of Programming, POPL'01*, pages 104–115, 2001.

[11] J. C. Godskesen. A calculus for mobile ad hoc networks. In *COORDINATION*, pages 132–150, 2007.

[12] J. C. Godskesen. A calculus for mobile ad-hoc networks with static location binding. *Electron. Notes Theor. Comput. Sci.*, 242(1):161–183, 2009.

[13] Y.-C. Hu and A. Perrig. A survey of secure wireless ad hoc routing. *IEEE Security and Privacy*, 2(3):28–39, 2004.

[14] Y.-C. Hu, A. Perrig, and D. B. Johnson. Ariadne: a secure on-demand routing protocol for ad hoc networks. *Wirel. Netw.*, 11(1-2):21–38, 2005.

[15] D. Johnson and D. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, 1996.

[16] J. D. Marshall, II, and X. Yuan. An analysis of the secure routing protocol for mobile ad hoc network route discovery: Using intuitive reasoning and formal verification to identify flaws. Technical report, THE FLORIDA STATE UNIVERSITY, 2003.

[17] P. Papadimitratos and Z. Haas. Secure routing for mobile ad hoc networks. In *Networks and Distributed Systems Modeling and Simulation*, 2002.

[18] A. Perrig, J. D. Tygar, D. Song, and R. Canetti. Efficient authentication and signing of multicast streams over lossy channels. *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 56, 2000.

[19] M. Poturalski, P. Papadimitratos, and J.-P. Hubaux. Towards provable secure neighbor discovery in wireless networks. *Proceedings of the 6th ACM workshop on Formal methods in security engineering*, pages 31–42, 2008.

[20] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[21] M. Saksena, O. Wibling, and B. Jonsson. Graph grammar modeling and verification of ad hoc routing protocols. 2008.

[22] A. Singh, C. R. Ramakrishnan, and S. A. Smolka. A process calculus for mobile ad hoc networks. *Sci. Comput. Program.*, 75(6):440–469, 2010.

[23] O. Wibling, J. Parrow, and A. Pears. Automatized verification of ad hoc routing protocols. *Formal Techniques for Networked and Distributed Systems FORTE*, pages 343–358, 2004.