Report

# Scyther
## Unbounded verification of security protocols

**Author(s):**
Cremers, C.J.F.

**Publication Date:**
2011

**Permanent Link:**

https://doi.org/10.3929/ethz-a-006809966 →

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report No. 572

# Scyther: Unbounded Verification of Security Protocols

C.J.F. Cremers
Information Security, ETH Zürich,
IFW C Haldeneggsteig 4
CH-8092 Zürich Switzerland.

**Abstract**

We present a new cryptographic protocol verification tool called Scyther. The tool is state-of-the-art in terms of verification speed and provides a number of novel features. (1) It can verify most protocols for an unbounded number of sessions in less than a second. Because no approximation methods are used, all attacks found are actual attacks on the model. (2) In cases where unbounded correctness cannot be determined, the algorithm functions as a classical bounded verification tool, and yields results for a bounded number of sessions. (3) Scyther can give a "complete characterization" of protocol roles, allowing protocol designers to spot unexpected possible behaviours early. (4) Contrary to most other verification tools, the user is not required to provide so-called *scenarios* for property verification, as all possible protocol behaviours are explored by default.
The algorithm expands on ideas from the Athena algorithm. We describe the algorithm, choice of heuristics, and discuss experimental results. The tool has been used already successfully for research as well as teaching of security protocol analysis.

Keywords: security, cryptographic protocols, formal methods, unbounded verification, authentication, secrecy.

## 1 Introduction

Many applications used today employ some form of network communication. For the majority of applications this means that there is communication over a network that is not secure, and which might be (partly) controlled by malicious parties. When secure communications are required, security protocols based on cryptography are used. Unfortunately, even when cryptography is assumed to be perfect, such security protocols are still often vulnerable to attacks. Such attacks exploit "features" of the protocols that were not considered by their designers. These behaviours are often missed because no formal verification is performed. Security protocols are usually fairly small, comprised of anything between three and twenty messages, and this seems to suggest that

their analysis is trivial. Unfortunately, this is not the case and can lure designers into a false sense of security, as protocol analysis is in fact a rather complex business.

The analysis of security protocols is complex, even for small protocols, because of the execution model and intruder model that is considered. First, the execution model implies that there is not just one session of the protocol, but rather that any number of agents are executing any number of sessions in parallel. Each instance of the protocol can generate new terms (such as random challenges, known as nonces), resulting in an unbounded number of terms. Second, the intruder model allows for malicious parties to intercept any traffic on the network, generate fresh data, and insert any messages that can be generated, either by analyzing received messages (including decryption) or by construction (including encryption). Furthermore, the malicious parties can compromise any number of communicating agents, which means they can even impersonate other agents. Still, we want to be sure that communicating agents have certain security guarantees.

During the last twenty years many models have been developed to understand protocol models and their behaviour. There is no consensus yet on which model is best suited in general. There is even less consensus on the tool side: there exist more security protocol analysis tools than security protocol models. The main reason for this seems to be that the security protocol models are underspecified, and tool designers have to make many decisions not strictly in the model. Each tool designer makes these decisions differently. The result is that there exists a variety of security protocol tools, that differ significantly not only in performance, but even more importantly, in underlying assumptions on the models, and implicit semantics of security properties.

As an ongoing research project, we attempt to address this situation in two ways. First, we have developed a protocol model that is as explicit as possible about execution assumptions and the exact formulations of security properties in [11,13,15]. In order to show the applicability of the developed theory, we have developed manual proofs and theoretical results in [14,21]. In parallel, we have developed an automatic protocol verification tool based on the protocol semantics model for the Dolev-Yao intruder model, and allows for verification of secrecy and authentication properties. The resulting tool, called Scyther, is presented in this paper. The tool is state-of-the-art in terms of performance and offers a number of novel features.

In this paper we sketch the protocol algorithm and present the novel features of the tool. A full description of the algorithm and complete proofs of all theorems can be found in [11]. The tool is freely available and can be downloaded at [9] for Windows, Linux and Mac OS X platforms. The tool distribution package includes a library of example protocols from the SPORE library.

Outline of the paper: We explain the protocol model and notion of patterns in Section 2. Then, we describe the verification algorithm in Section 3. We discuss various features of the tool in Section 4. The underlying verification algorithm of the tool is related to the algorithm presented in [23]. This issue is discussed in more detail in Section 5, along with other related work. We conclude in Section 6 and discuss future work. In the appendices we show an example input file and give an example of complete characterization of the Needham-Schroeder protocol.

## 2 Security Protocols and patterns

As the protocol model underlying our method we choose the operational semantics as defined in [11,13]. We first give a brief overview highlighting the most important concepts. Then we show how all possible protocol executions can be captured in terms of trace classes.

## 2.1 Protocol model

A protocol specification defines the exchange of message terms between roles. We call these terms used in the specification "role terms" to distinguish them from the actual terms occurring during the execution of a protocol. We construct Role terms using the sets *Var*, denoting variables that are used to store received messages, *Const*, denoting constants which are freshly generated for each instantiation of a role, and are therefore considered local constants, *Role*, denoting roles, and *Func*, denoting function names. The set of *Role Terms* is defined as the basic term sets, extended with constructors for pairing, encryption, and function application, denoted respectively by $(\_,\_)$, $\{\!|\,\_\,|\!\}_\_$, and $Func(\_)$:

$$\begin{aligned} RoleTerm \quad ::= \quad & Var \mid Const \mid Role \mid Func \\ & \mid (RoleTerm, RoleTerm) \mid \{\!|\, RoleTerm \,|\!\}_{RoleTerm} \mid Func(RoleTerm) \end{aligned}$$

Functions are considered to be global, and have an arity which must be respected in all terms according to their definition. If global constants occur in a protocol, we model them as functions of arity zero.

Encrypted terms can only be decrypted by either the same term (for symmetric encryption) or the inverse key (for asymmetric encryption). We define a function that yields the inverse for any role term: $\_^{-1} : RoleTerm \to RoleTerm$. All terms $rt$ for which we have $rt^{-1} = rt$ are considered to be symmetric keys. We explicitly allow for composed keys.

We describe protocols as sets of roles, which contain role events. We define the set of events *RoleEvent* using two new sets, labels *Label* and security claims *Claim*.

$$\begin{aligned} RoleEvent \quad = \quad & \big\{ send_\ell(R, R', rt), read_\ell(R', R, rt), claim_\ell(R, c, rt) \\ & \big|\ \ell \in Label, R, R' \in Role, rt \in RoleTerm, c \in Claim \big\} \end{aligned}$$

Event $send_\ell(R, R', rt)$ denotes the sending of message $rt$ by role $R$, intended for role $R'$. Likewise, $read_\ell(R', R, rt)$ denotes the reception of message $rt$ by role $R'$, apparently sent by role $R$. Event $claim_\ell(R, c, rt)$ expresses that $R$ upon execution of this event expects that security goal $c$ holds with optional parameter $rt$. A claim event denotes a *local security claim*, which means that it only concerns role $R$ and does not express any expectations at other roles. The labels $\ell$ that tag the events are used to disambiguate similar occurrences of the same event in a protocol specification. A second use of these labels will be to express the relation between corresponding send and read events, which is often not expressed in role-based protocol descriptions.

A role specification consists of a list of role events. We write $[\varepsilon 1, \varepsilon 2]$ to denote the list of two events $\varepsilon 1$ and $\varepsilon 2$. A protocol specifies the behaviour of a number of roles by means of a partial function from the set *Role* to $RoleEvent^*$. Thus, each role description corresponds to a (totally ordered) list of events. For a role $R$ this ordering is denoted by $\prec_R$.

**Example 1** (Protocol description)**.** *Consider the following example protocol, for which the informal protocol description is the following:*

$$i \to r : \{\!|\, \{\!|\, ni \,|\!\}_{pk(r)} \,|\!\}_{sk(i)}$$

*It contains two roles. The initiator sends to the responder a freshly generated value $ni$, encrypts it with the public key of the responder, and signs it using its secret key. Thus, we get the following*

*protocol definition:*

$$\begin{array}{ll} signprot(i) = & [\ send_1(i, r, \{\!|\ \{\!|\ ni\ |\!\}_{pk(r)}\ |\!\}_{sk(i)})\ ] \\ signprot(r) = & [\ read_1(i, r, \{\!|\ \{\!|\ V\ |\!\}_{pk(r)}\ |\!\}_{sk(i)})\ ,\ claim(r, secret, V)\ ] \end{array}$$

## 2.2 Protocol execution as patterns

We interpret protocol actions as in [13]. The semantics of a protocol description in the presence of an intruder is defined as the set of possible execution traces. A trace is a totally ordered set of events, in which no variables occur. One can also express protocol behaviour in terms of classes of traces, by using representations which require no total order on the events, and in which variables may occur. This approach is similar to the Strand Spaces approach of [26] or the Cord Spaces [17].

Each role in a protocol description can be executed any number of times, by an unbounded number of agents. We refer to a unique execution of a role as a *run*. A run might be only a partial execution of a role. Thus, a run is some specific instance of a (partial) role by an agent.[1] We identify each separate run by its run identifier: this is a unique identifier at the meta level, and used distinguish between different runs.

We exploit the run identifier to capture the notion of terms that are local to a run at a syntactical level. Each run (with associated run identifier) can have unique variables and local uniquely generated values such as nonces. We use the notation $V\sharp\theta$ and $ni\sharp\theta$ to denote respectively the variable $V$ of run $\theta$, and the nonce $ni$ generated by run $\theta$. Similarly, $R\sharp\theta$ denotes the agent assigned to the $R$ role in run $\theta$. Thus, for a run with identifier $\theta$, the role term $rt$ is turned into a run term by appending the run identifier to all subterms that are local values, by the following inductive definition:

$$RunInst(\theta, rt) = \begin{cases} rt\sharp\theta & \text{if } rt \in Var \cup Const \cup Role \\ (RunInst(\theta, rt1), RunInst(\theta, rt2)) & \text{if } rt \equiv (rt1, rt2) \\ \{\!|\ RunInst(\theta, rt1)\ |\!\}_{RunInst(\theta, rt2)} & \text{if } rt \equiv \{\!|\ rt1\ |\!\}_{rt2} \\ f(RunInst(\theta, rt1), \ldots, RunInst(\theta, rtn)) & \text{if } rt \equiv f(rt1, \ldots, rtn) \end{cases}$$

In such run terms, role names and local variables are considered as variables, and all other terms are constants. Thus, a run term $(i\sharp\theta, r\sharp\theta, ni\sharp\theta)$ represents a class of run terms, of which $(A, A, ni\sharp\theta)$, $(A, B, ni\sharp\theta)$ etc., are members.

When instantiated in a run, Role events are labeled in the same way. For a run identifier $\theta$, role event $\varepsilon$, role term $rt$ and label $\ell$, we inductively define the corresponding run event as

$$RunInst(\theta, \varepsilon) = \begin{cases} send_\ell(RunInst(\theta, rt))\sharp\theta & \text{if } \varepsilon = send_\ell(rt) \\ read_\ell(RunInst(\theta, rt))\sharp\theta & \text{if } \varepsilon = read_\ell(rt) \\ claim_\ell(RunInst(\theta, rt))\sharp\theta & \text{if } \varepsilon = claim_\ell(rt) \end{cases}$$

**Example 2** (Run events)**.** *For the protocol in Example 1 the following are run events:*

$$send_1(i\sharp\theta, r\sharp\theta, \{\!|\ \{\!|\ ni\sharp\theta\ |\!\}_{pk(r\sharp\theta)}\ |\!\}_{sk(i\sharp\theta)})\sharp\theta \qquad read_1(i\sharp\theta', r\sharp\theta', \{\!|\ \{\!|\ V\sharp\theta'\ |\!\}_{pk(r\sharp\theta')}\ |\!\}_{sk(i\sharp\theta')})\sharp\theta'$$

*In this example, $V$ and all the role names are considered variables, with an associated type. The role names are of type Agent, and $V$ is of type Nonce. The variables may be substituted by any term that matches the type. In the tool input language variables and constants are explicitly typed, as can be seen in Appendix A.*

[1] In other formalisms, what we call a run is sometimes known as a "thread", "process", or "regular strand".

We introduce three helper functions. The function *roleevent*() yields the role event from which a run event was constructed. (The labels of the events ensure that there is a unique role event.) The function *role*() yields the role of a role event or run event. The function *runid*() yields the run identifier of a run event.

Next we introduce the concept of a *trace pattern*, which is a partially ordered, symbolic representation of a set of traces.

**Definition 3** (Trace pattern)**.** *Let Subst denote all possible substitutions of constants, roles or variables by run terms. A tuple* $(E, \rightarrow)$ *is a trace pattern iff* $E \subseteq RoleEvent$ *and*

$$\forall e, e' \in E : \big(runid(e) = runid(e') \wedge roleevent(e) = roleevent(e')\big) \Rightarrow e = e'$$

*Then, for role consistency we have that*

$$\forall e, e' \in E : runid(e) = runid(e') \Rightarrow \big(role(e) = role(e') \wedge \exists \sigma \in Subst \rightarrow RunTerm : \sigma(roleevent(e)) = e \wedge \sigma(roleevent(e')) = e'\big)$$

*and we have that the* $\rightarrow$ *relation connects events from E, forming a directed acyclic graph.*

The first requirement states that each role event can only occur once in a role. The second requirement expresses that the pattern is consistent with regard to the role execution of the protocol. Once a variable is instantiated in the run, it must be similarly instantiated in the other events of the same run.

We use $\rightarrow^*$ to denote the smallest transitive relation containing $\rightarrow$.

**Example 4** (Trace pattern)**.** *Consider the following run events that are instantiations of of the events of the signprot protocol of Example 1.*

$$e1 = send_1(A, r\sharp\theta, \{\!|\, \{\!|\, ni\sharp\theta \,|\!\}_{pk(r\sharp\theta)} \,|\!\}_{sk(A)})\sharp\theta$$
$$e2 = read_1(i\sharp\theta', B, \{\!|\, \{\!|\, ni\sharp\theta \,|\!\}_{pk(B)} \,|\!\}_{sk(i\sharp\theta')})\sharp\theta'$$

*Then* $(\{e1, e2\}, e1 \rightarrow e2)$ *is a trace pattern. The interpretation of this pattern is that* $e1$ *is part of a run of the agent A, and* $e2$ *is part of a run of B. A generates a nonce* $ni\sharp\theta$*, which is later received by B.*

*We will later see that this pattern can not occur for just any substitution of the variables* $i\sharp\theta'$ *and* $r\sharp\theta$*: as the nonce is kept secret and messages are signed, the protocol enforces here that the pattern can only occur if* $i\sharp\theta' = A$ *and* $r\sharp\theta = B$*.*

The question now becomes: Suppose we have a certain pattern, can we construct a trace that contains it, by reasoning backwards?

## 2.3 Explicit trace patterns for reasoning about patterns

Of most trace patterns (like the one in the previous example) it is not immediately clear whether they can occur in a trace. In order to capture this form of reasoning shown the example, we introduce the notion of *explicit trace pattern*. Such a pattern makes parts of the intruder derivations explicit, wherever they are needed for the backwards reasoning approach.

Explicit trace patterns serve two purposes. One, they enable backwards reasoning about intruder inferences involving encryption and decryption. Second, they allow us to easily deduce that the

pattern can occur in a trace of a protocol. In fact, we can define all possible protocol behaviours in terms of explicit patterns: the protocol semantics are then defined as the set of possible patterns that can occur during execution[2].

For all terms we define the function $parts : RunTerm \to \mathcal{P}(RunTerm)$ such that

$$parts(rt) = \begin{cases} parts(rt1) \cup parts(rt2) & \text{if } rt = (rt1, rt2) \\ \emptyset & \text{if } rt = V\sharp\theta \\ \{rt\} & \text{otherwise} \end{cases}$$

We consider all non-tuple terms that are not (uninstantiated) variables to be a *part* of the term. Uninstantiated variables are omitted as the initial intruder knowledge contains intruder constants of all types (denoted by *IntruderConst*), corresponding to the intuition that an intruder can generate fresh values of any type.

We extend the set of run events by two additional events *encr* and *decr* that explicitly capture the intruders actions of encryption and decryption.

For all labels $\ell$ and terms $rt1, rt2, rt3$, the parts that are considered the input and output of an event $e$ are defined as $in, out : RunEvent \to \mathcal{P}(RunTerm)$, where

$$in(e) = \begin{cases} parts(rt1) & \text{if } e = read_\ell(rt1) \\ parts((rt1, rt2)) & \text{if } e = encr(rt1, rt2) \\ \{\{\!|\, rt1 \,|\!\}_{rt2}\} \cup parts(rt2^{-1}) & \text{if } e = decr(\{\!|\, rt1 \,|\!\}_{rt2}, rt2^{-1}) \\ \emptyset & \text{otherwise} \end{cases}$$

$$out(e) = \begin{cases} parts(rt1) & \text{if } e = send_\ell(rt1) \\ \{\{\!|\, rt1 \,|\!\}_{rt2}\} & \text{if } e = encr(rt1, rt2) \\ parts(rt1) & \text{if } e = decr(\{\!|\, rt1 \,|\!\}_{rt2}, rt2^{-1}) \\ \emptyset & \text{otherwise} \end{cases}$$

Recall that $\prec_r$ denotes the order of the events within a role.

**Definition 5** (Explicit trace pattern)**.** *Let $(E, \to)$ be a trace pattern. We call $(E, \to)$ an explicit trace pattern iff for all $e \in E$*

$$\Big(\exists re \prec_r (roleevent(e)) \Rightarrow \big(\exists e' \in E : roleevent(e') = re \wedge runid(e) = runid(e')\big)\Big)$$
$$\wedge \Big(\forall t \in in(e) : \exists e' \in E : t \in out(e') \wedge e' \xrightarrow{t} e \wedge \big(\neg\exists e'' : e'' \to^* e' \wedge t \in out(e'')\big)\Big)$$

The first conjunct expresses that runs are executions of (partial) roles: if a role event of a run is preceded by another, this must also be present in the pattern. The second conjunct ensures that all *in* relations are satisfied, e. g. that all read events can be triggered on the basis of previous knowledge. Furthermore we require that the $\xrightarrow{t}$ relation is minimal, i. e. there must be no earlier occurrences of the term $t$.

Three examples of explicit trace patterns (with additional annotations) can be found in Figures 1 and 2 in the appendix. Explicit trace patterns are closely related to the notions of bundles in the Strand Spaces model, although here there is a much more clear relation between the pattern and the protocol, and fresh values are automatically dealt with by means of the local constants construction.

[2]The Strand Spaces model captures this notion in terms of bundles: bundles are effectively patterns that can occur during the execution of a protocol.

**Theorem 6** (Explicit trace patterns correspond to trace of the protocol)**.** *Let pt be an explicit trace pattern of a protocol $P$. Then there exists an execution history of the protocol in which the pattern occurs.*

*Proof sketch.* Because the explicit pattern is non-cyclic and the intruder can generate fresh terms of any type, we can refine the order into a total order, substitute all variables by intruder generated constants, omitting the encrypt and decrypt events. This results in a execution history of the protocol that matches the operational semantics from [13]. Because the requirements on the explicit trace pattern, all messages occurring in read events in the resulting behaviour can be inferred by the intruder from previous messages.

**Theorem 7** (All execution histories of a protocol can be expressed using explicit patterns)**.** *Let $\alpha$ be a execution history of a protocol $P$. Then we can construct an explicit trace pattern of which $\alpha$ is a possible linearization.*

*Proof sketch.* Given an execution history, the corresponding pattern contains the events of the trace and additional encrypt and decrypt events. However, it contains less order and some order relations are labeled. The proof uses only induction on the length of $\alpha$, by showing how to construct the partial orderings between the events of the pattern. We distinguish two types: unlabeled orderings are induced by the order of events in a run, and labeled orderings are induced by the knowledge inference of the intruder.

# 3 Verification algorithm

Starting from a (non-explicit) trace pattern, we enumerate all possible ways in which all the read, encrypt, and decrypt events in the pattern can be provided with a suitably labeled incoming arrow, ultimately to arrive at an explicit pattern.

If in any branch of the enumeration process we find that all events are enabled (i. e. have suitably labeled incoming edges), we have an explicit trace class, representing a class of traces in which the pattern occurs. Conversely, if no branch of the enumeration process yields an explicit pattern, then the original pattern will never occur in any trace of the protocol. Thus, the enumeration process constructs the set of all possible explicit patterns that include the starting pattern.

This sketched algorithm is not guaranteed to terminate, but by introducing a parameter $mr$, the algorithm in the next section is guaranteed to terminate whilst retaining the possibility of unbounded verification by generation of the full set (as one would get without enforcing termination).

Intuitively, the verification method starts from the pattern and tries to establish whether this can be reached from the initial state of the system by applying the protocol rules. This is similar to backwards searches in generic model checking tools, such as e. g. [18]. However, we exploit the specifics of the security protocol problem as much as possible, in order to achieve a more efficient procedure.

The ingredients of the algorithm are the following. Given a pattern $pt$ of a protocol $P$, the algorithm returns a set of explicit patterns.

(i) If $pt$ cannot be refined into an explicit pattern, return the empty set. Possible causes are: the ordering on $pt$ is not minimal or cyclic.

(ii) If $pt$ is an explicit pattern, return $\{pt\}$.

(iii) $pt$ is not realizable, i. e. not all events are enabled.

*Choose* an event $e \in pt$ which is not enabled. In other words: there is no "source" for a term $t$ which is part of $e$. This choice corresponds to a heuristic.

Apply case distinction on all possible sources of $t$. In each case do the following:

(a) Extend the pattern if needed with new events

(b) Instantiate variables if needed

(c) Iterate the algorithm at (i) for the resulting refined pattern $pt'$

For verification, security properties are encoded as patterns. In case of a secrecy claim, an additional read of the supposedly secret term is added to the pattern. If an explicit pattern exist that contains this pattern, the secrecy claim is false and the explicit pattern represents an attack. For authentication properties, we generate all explicit patterns that include the authentication claim. For these explicit patterns we can conclude whether the authentication property holds, e. g. if the pattern includes events of a suitable partner in the desired order, or whether they constitute a counterexample.

## 3.1 Algorithm

To work out the details of the algorithm sketched before, we need some further machinery. Note that the full details of the algorithm and correctness proofs can be found in [11].

**Definition 8** (Most general unifier $MGU$)**.** *Let $\phi$ be a substitution of uninstantiated variables by run terms. We call $\phi$ a unifier of term $t1$ and term $t1$ if $\phi(t1) = \phi(t2)$. We call $\phi$ the most general unifier of two terms $t1, t2$, notation $\phi = MGU(t1, t2)$, if for any other unifier $\phi'$ there exists a substitution $\phi''$, such that $\phi' = \phi \circ \phi''$.*

The above definition assumes variables are typeless. In the remainder, we will assume that $MGU$ is defined in such a way that the type check constraints on variables are met.
We generalize the notion of unification to so-called decryption unification, which captures all the ways in which a term $t1$ can be unified with a (sub)term of another term $t2$ (possibly after repeated decryption and projection operations), and the terms that need to be decrypted in order to extract this unified term from $t2$. We write $[\,]$ to denote the empty list.

**Definition 9** (most general decryption unifier $MGDU$)**.** *Let $\phi$ be a substitution of uninstantiated variables by run terms. We call $(\phi, L)$ a decryption unifier of a term $t1$ and a term $t2$, notation $(\phi, L) \in DU(t1, t2)$, if either*

- $L = [\,] \ \wedge \ \phi(t1) \in parts(\phi(t2))$, *or*
- $L = L' \cdot \{\!|\, m \,|\!\}_k$, $\{\!|\, m \,|\!\}_k \in parts(\phi(t2)) \ \wedge \ (\phi, L') \in DU(t1, m)$.

*We call a set of decryption unifiers $S$ the most general decryption unifiers of $t1$,$t2$, notation $S = MGDU(t1, t2)$, if and only if*

- *for all $(\phi, L) \in S$ we have that $(\phi, L) \in DU(t1, t2)$, and*
- *for any decryption unifier $(\phi, L) \in DU(t1, t2)$, there exists a decryption unifier $(\phi', L') \in MGDU$ and a substitution $\phi''$, such that $\phi' = \phi \circ \phi''$.*

The set $MGDU$ captures all ways in which a term $t1$ can be the result of applying (repeated) decryptions and projections to $t2$. Each element of the set consists of a unifying substitution, and a list of terms that need to be decrypted. Each such tuple uniquely defines a decryption sequence that starts with the sending of $t2$, and results in $t1$ becoming known to the intruder.

**Example 10.** *Let $V$ and $W$ be variables, such their sets of allowed substitutions $type(V)$,$type(W)$ contain only basic terms, and thus no tuples or encryptions. Then we have that*

$$MGDU(ni\sharp 1, (V\sharp 1, \{\!|\, W\sharp 2 \,|\!\}_{pk(r\sharp 2)})) = \Big\{ \quad \big(\{V\sharp 1 \mapsto ni\sharp 1\} \quad , [\,]\big), \\ \big(\{W\sharp 2 \mapsto ni\sharp 1\} \quad , [\{\!|\, W\sharp 2 \,|\!\}_{pk(r\sharp 2)}]\big) \Big\}$$

**Lemma 11** (Conditions under which the $MGDU$ set is finite)**.** *Given two terms $t1, t2$, such that for each variable $V$ occurring in $t1$ and $t2$, $type(V)$ contains no tuples and encryptions, the set $MGDU(t1, t2)$ is finite.*

*Proof sketch.* Let $t1$ and $t2$ be terms. We use the following procedure to compute the set of most general decryption unifiers:

$$MGDU(t1, t2) = \quad \big\{(\phi, [\,]) \;\big|\; t' \in parts(t2) \wedge \phi = MGU(t1, t')\big\} \cup \\ \big\{(\phi, L \cdot [\{\!|\, m \,|\!\}_k]) \;\big|\; \{\!|\, m \,|\!\}_k \in parts(t2) \wedge (\phi, L) \in MGDU(t1, m))\big\}$$

The first half of the union is a finite set. The second component is defined recursively. $t2$ includes finitely many applications of the encryption operator, and each subterm contains finitely many parts. As any subsequent substitutions do no change the number of tuples or encryptions, the iteration is guaranteed to terminate.

On the other hand, if $t2$ contains variables which can be instantiated with tuples or encryptions, the set $MGDU$ is not guaranteed to be finite.

Using these notions, we describe Algorithm 1 on the following page, where we still need to explain *selectOpen*, *chainToExisting* and *chainToNew*. *selectOpen* denotes a heuristic, explained in the next section. Both other functions return refinements of a pattern, in which the given chain (a result of $MGDU$) is added to and connected to events in the pattern. In case of *chainToExisting*, the chain is connected to an event already existing in the pattern. In case of *chainToNew*, the chain is connected to an event that is added to the pattern. This corresponds to receiving a term from a run not yet in the pattern, and generates a fresh run identifier. Observe that introducing a new run of a role requires also introducing the preceding events of that role, thus possible introducing new read events into the pattern.

**Theorem 12** (Termination)**.** *Algorithm 1 terminates.*

*Proof sketch.* Each iteration either decreases the number of unbound goals or increases the number of runs. From these two elements we can construct a non-increasing invariant that ensures termination.

If the result of the algorithm is the empty set, the pattern does not occur at all in any possible execution history of the protocol. In the pattern represented an attack, this means that the attack is not possible on the protocol.

If the result of the algorithm does not contain *NotComplete*, all possible explicit patterns have been found. If the result contains *NotComplete*, all explicit patterns that contain at most $mr$ runs have been determined, but there might still be explicit patterns that contain more runs. This last answer corresponds to a traditional bounded model-checking result, and means that all traces containing $mr$ runs have been considered, but there might be traces that contain more runs.

**Algorithm 1** ExplicitPatterns(pt,mr)

**Require:** $pt$ is a trace pattern, and $mr$ is an integer.

**Ensure:** Returns a set of explicit patterns. The set may include the special symbol '*NotComplete*' when the full set of explicit patterns can not be determined.

**if** $runs(pt) > mr$ **then**
  return $\{NotComplete\}$
**else**
  **if** $isCyclic(pt) \vee notMinimal(pt)$ **then**
    return $\emptyset$
  **else**
    **if** $explicit(pt)$ **then**
      return $\{pt\}$
    **else**
      $(e, t) = selectOpen(pt)$
      $outTerms = \{t' \mid t' = \{\!|\, t1 \,|\!\}_{t2} \vee (re \in RoleEvent \wedge t' \in out(re))\}$
      $result = \emptyset$
      **for all** $chain \in \{MGDU(t', t) \mid t' \in outTerms\}$ **do**
        $result = result \cup$ ExplicitPatterns$(chainToExisting(pt, e, t, chain), mr)$
        $result = result \cup$ ExplicitPatterns$(chainToNew(pt, e, t, chain), mr)$
      **end for**
      return $result$
    **end if**
  **end if**
**end if**

## 3.2 Determining heuristics

For the verification algorithm, there are two remaining factors to be determined: the heuristic *selectOpen* that guides the search, and the default $mr$ setting on the number of runs. We have investigated possible options extensively and report our results in [11]. Here we summarize the results.

**Heuristic** The verification algorithm includes an important heuristic. Given that all read events must be enabled, we can select one of the events that is not enabled yet. This is done by a heuristic. The heuristic can influence the number of states traversed, but also the maximum size of the set of pattern events at which contradictions are found. This means the heuristic is important not only for the speed of the verification, but also for improving the number of cases in which verification is complete when the algorithm is invoked with a specific choice for the parameter $mr$. In [11] we investigate several heuristics and compare their effect on the verification process for the test set of over 200 protocols. Based on these tests, we find that the most effective heuristic is a combination of the following three simpler heuristics, combined by lexicographical ordering, i. e. on (PrivateKeys,Constants,Decryptions).

**PrivateKeys** Give priority to goals that contain a long-term private key[3] as a subterm; next, give priority to goals that contain a public key. All other terms have lower priority.

**Constants** For each open goal term $rt$, the number of local constants that are a subterm of $rt$, is

[3] Observe that the semantics do not explicitly mention concepts such as 'long-term private' or 'public' key, (there might be no such terms, or multiple key infrastructures). However we can determine these using a simple heuristic.

divided by the number of basic terms that are a subterm of $rt$. The goal with the highest ratio is selected.

**Decryptions** Open goals that correspond to the keys needed for decrypt events are given higher priority, unless these keys are in the initial intruder knowledge.

### 3.3 Choosing the parameter $mr$

For the over 200 protocols analyzed for this work, we did not find any attacks that involved more than $x + 1$ runs, where $x$ is the number of roles of the protocol, except for the $f^N g^N$ family of protocols from [19]. The exceptions involve a family of protocols that was specifically tailored to be correct for $N$ runs, but incorrect for $N + 1$ runs. This indicates that for practical purposes, initial verification with three or four runs would be sufficient.

Because increasing $mr$ can improve the rate of complete characterization, but also increases verification time, there is an inherent trade-off between completeness and verification time. We have determined that the optimal default setting for the parameter $mr$ is five runs, at which we achieve a decision for 82% of the protocols. In the remaining 18% we achieve a bounded result similar to other model checking methods. Note that the Scyther tool allows the user to change (or remove) the default setting of five runs. As an extreme example, we have found an attack that involves 51 runs (on the $f^{50}g^{50}$ protocol from [19]) in 137 seconds.

## 4 Scyther tool features

Scyther can be used in three ways: as a command-line tool, as a backend for analysis programs using interface functions in the Python language, or by using the graphical user interface. The full tool set is available for Windows, Linux, and Max OS X (both Intel and PPC) platforms. Scyther can be freely downloaded at [9], and comes with a library of example protocols modeled after the SPORE library [25](a continuation of the Clark Jacob library [5]).

### 4.1 Performance

Comparing protocol tools is non-trivial as tools have many subtle underlying differences, not only in features but also in the state spaces they explore, and is therefore beyond the scope of this paper. We have recently addressed this situation and have done a state space analysis and performance comparison of several protocol verification tools, including Scyther, in [12]. Here we restrict ourselves to highlighting some simple cases, performed on an AMD Sempron 3000 with 1GB of ram running Linux, by using the command-line.

**Needham-Schroeder public key (total time: 0.04s)** Scyther performs unbounded verification the secrecy and authentication claims for the initiator role, and finds attacks against all claims of the responder.

**Needham-Schroeder-Lowe (total time: 0.02s)** Scyther performs unbounded verification the secrecy and authentication claims of both roles.

**TLS (total time: 0.12s)** This is an eight-message protocol modeled as in the AVISPA library. Scyther performs unbounded verification for all claims except for the authentication claim of the $a$ role, where it finds a reflection attack, not modeled in the AVISPA scenario.[4]

[4]Note that this is a modeling problem in the AVISPA library, as this attack does not occur against e. g. Paulson's

### 4.2 Complete characterization

Given a specific protocol event, there are usually only a few different ways in which it can occur in a trace. From the point of view the protocol designer, when a certain event occurs, all preceding events in the protocol description should have also occurred, reflecting the "correct" or "intended" behaviour of the protocol. In case of an attack, there is typically an alternative way to reach a certain event, unforeseen by the protocol designer, such as the man-in-the-middle attack for Needham-Schroeder. In either case, there are only a few distinct behaviours possible. Determining all possible behaviours leading up to a certain protocol event is referred to as *complete characterization*. We owe this terminology to the (independently developed) work of [16], in which a different approach is taken to arrive at such characterizations.

**Definition 13** (Complete characterization)**.** *Let pt be a trace pattern, typically including only a single role of a protocol. Let* $S = \text{ExplicitPatterns}(pt, mr)$ *for some* $mr$*. We have that* $S$ *is a complete characterization of pt if and only if NotComplete* $\notin S$*.*

Essentially, a complete characterization of a protocol role is the full set of possible explicit patterns that include the role. In Appendix A we show a complete characterization of the roles of the Needham-Schroeder protocol.

## 5 Related work

There exists a wide range of automatic security protocol verification tools. Scyther improves on most approaches by its speed and features, but there are some tool that provide features not currently available in Scyther. As stated in Section 4.1, a full performance analysis can be found in [12]. Here we briefly discuss some prominent approaches that are currently freely available.

**CoProVe:** Based on [20], in which verification in the Strand Spaces model is translated into a constraint solving problem, a more efficient constraint solving method was developed in [7]. The method uses constraint solving, optimized for protocol analysis, and a minimal form of partial order reduction, similar to the one used in [6]. A second version of this tool does not use partial order reduction, enabling it to verify properties of the logic PS-LTL [8].

**AVISPA:** The AVISPA tool set [2] comprises a common high-level input language (HLPSL) and four verification backends: OFMC, SATMC, CL-ATSE and TA4SP. For bounded verification of protocols (using small bounds similar to three runs), AVISPA is state-of-the-art in terms of both speed and features. As an example of tools offering features Scyther does not, we mention that some of the backends support user-defined algebraic theories, allowing for correct modeling of e. g. exclusive-or, or Diffie-Hellman exponentiation. Unbounded verification using TA4SP is still in an early stage, and only a handful of protocols can be verified.

**ProVerif:** In ProVerif [4], protocol steps are represented by Horn clauses. The system can efficiently handle an unbounded number of sessions of the protocol by using overabstraction on the nonces. As a consequence, when the system claims that the protocol preserves the secrecy of some value, this is correct; however it can generate false attacks too. If a false attack is detected, no statement about the correctness of the property can be made. The algorithm is not guaranteed to terminate.

modeling of TLS.

**Athena:** Although Athena is not freely available, we discuss it here because of its relation to the algorithm used here. The algorithm described in [23] served as a starting point for the development of the algorithm used in Scyther. When terminating, the algorithm provides either a counterexample if the formula under examination is false, or establishes a claimed proof that the formula is true. Alternatively, Athena can be used with a bound (e. g. on the number of runs), in which case termination is guaranteed, but it can guarantee at best that there exist no attacks within the bound.

Due to changes in the theoretical foundations, we improve on the basic Athena algorithm described in at the following ways. First, our algorithm is guaranteed to terminate whilst still giving complete characterizations for the vast majority of protocols (i. e. deciding security properties for an unbounded number of sessions). Second, we extend the method to our operational semantics, allowing for e.g. multiple key structures and non-atomic keys. As a side result, security properties are defined as claim events, and there is no need to set up complex scenarios for the verification of properties that are error-prone. Third, we revise the algorithm to yield complete characterizations. This allows us to verify a larger class of security properties, including ordering-based security properties such as synchronisation. Fourth, we have analysed the heuristics in the algorithm and their impact on the verification process, resulting in our choice of heuristics.

Furthermore, there are some technical problems in the Athena theory that cast doubt over the claim of unbounded verification proofs, briefly described in Appendix B.

# 6 Conclusions and future work

We have presented Scyther, a new automatic protocol verification tool. Scyther can be used for finding attacks or performing unbounded verification. For both purposes Scyther compares well to other protocol analysis tools. It combines desirable features of model checking methods (finding attacks, termination) and theorem proving or abstraction-based methods (unbounded verification). Scyther additionally has a number of novel features such as complete characterization and attack selection, which are not offered by other tools.

Exploiting the state-of-the-art performance of Scyther, we have discovered previously unknown multi-protocol attacks reported in [10]. We have discovered previously unknown attacks on the synchronisation of two multi-party authentication protocols in [14]. Scyther has also been used to verify theoretical results regarding protocol composition in [1]. In short, Scyther has been successfully used for the verification and design of protocols, as well as for supporting theoretical research.

Furthermore, Scyther is currently being used not only for research but also for teaching purposes at several universities. For teaching purposes, the clear relation between the protocol specification and the protocol semantics have proven useful in explaining the fundamentals of protocol design and analysis. The concise protocol descriptions help in focussing on the protocol as opposed to tool details, in contrast to other protocol tools that require the specification of error-prone scenarios for the verification of properties.

Currently Scyther supports symmetric and asymmetric encryption, and hash functions. In future work we will investigate the incorporation of Diffie-Hellman and exclusive-or primitives.

# References


[1] S. Andova, C.J.F. Cremers, K. Gjøsteen, S. Mauw, S.F. Mjølsnes, and S. Radomirović. Sufficient conditions for composing security protocols, 2006. In preparation.

[2] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, L. Cuellar, P.H. Drielsma, P. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proc. Computer Aided Verification'05 (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

[3] S. Berezin. Extensions to Athena: Constraint satisfiability problem and new pruning theorems based on type system extensions for messages. `http://www.sergeyberezin.com/papers/athena-extensions.ps`, 2001.

[4] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96, Cape Breton, June 2001. IEEE Computer Society.

[5] J.A. Clark and J.L. Jacob. A survey of authentication protocol literature. `http://citeseer.ist.psu.edu/clark97survey.html`, 1997.

[6] E.M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACMTSEM: ACM Transactions on Software Engineering and Methodology*, 9(4):443–487, 2000.

[7] R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. In M.V. Hermenegildo and G. Puebla, editors, *Proc. 9th International Static Analysis Symposium (SAS)*, volume 2477 of *Lecture Notes in Computer Science*, pages 326–341, Spain, Sep 2002. Springer.

[8] R.J. Corin, A. Saptawijaya, and S. Etalle. A logic for constraint-based security protocol analysis. In *2006 IEEE Symposium on Security and Privacy*, Los Alamitos, California, 2006. IEEE Computer Society.

[9] C.J.F. Cremers. The Scyther tool: Automatic verification of security protocols. `http://people.inf.ethz.ch/cremersc/scyther/index.html`.

[10] C.J.F. Cremers. Feasibility of multi-protocol attacks. In *Proc. of The First International Conference on Availability, Reliability and Security (ARES)*, pages 287–294, Vienna, Austria, April 2006. IEEE Computer Society.

[11] C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, 2006.

[12] C.J.F. Cremers and P. Lafourcade. Comparing state spaces in security protocol verification, 2007.

[13] C.J.F. Cremers and S. Mauw. Operational semantics of security protocols. In S. Leue and T. Systä, editors, *Scenarios: Models, Transformations and Tools, International Workshop, Dagstuhl Castle, Germany, September 7-12, 2003, Revised Selected Papers*, volume 3466 of *Lecture Notes in Computer Science*. Springer, 2005.

[14] C.J.F. Cremers and S. Mauw. Generalizing Needham-Schroeder-Lowe for multi-party authentication, 2006. Computer Science Report CSR 06-04, Eindhoven University of Technology.

[15] C.J.F. Cremers, S. Mauw, and E.P. de Vink. Injective synchronisation: an extension of the authentication hierarchy. *Theoretical Computer Science*, 2006.

[16] S. Dogmi, J.D. Guttman, and F.J. Thayer. Skeletons and the shapes of bundles. `http://www.ccs.neu.edu/home/guttman/skeletons.pdf`, 2006.

[17] N.A. Durgin, J.C. Mitchell, and D. Pavlovic. A compositional logic for protocol correctness. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 241–272, 2001.

[18] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[19] J.K. Millen. A necessarily parallel attack. In N. Heintze and E. Clarke, editors, *Workshop on Formal Methods and Security Protocols*, Trento, Italy, 1999.

[20] J.K. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. 8th ACM Conference on Computer and Communications Security*, pages 166–175. ACM Press, 2001.

[21] N. Palm. Bewijzen van security protocollen in een trace model. Master's thesis, Technische Universiteit Eindhoven, Department of Mathematics and Computer Science, 2004.

[22] D. Song. Athena: a new efficient automatic checker for security protocol analysis. In *Proc. 12th IEEE Computer Security Foundations Workshop (CSFW)*, pages 192–202. IEEE Computer Society, 1999.

[23] D. Song. *An Automatic Approach for Building Secure Systems.* PhD thesis, UC Berkeley, December 2003.

[24] D. Song, S. Berezin, and A. Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.

[25] Security protocols open repository (SPORE). `http://www.lsv.ens-cachan.fr/spore`.

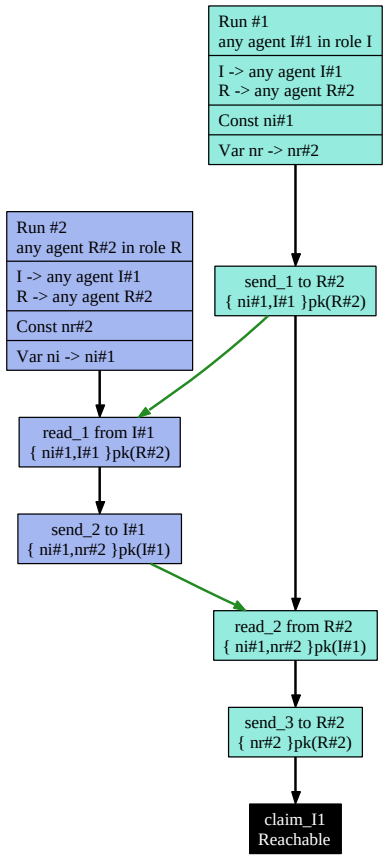
[26] F.J. Thayer, J.C. Herzog, and J.D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.

# A Example input file and complete characterization

The input below is an example of a file describing the Needham-Schroeder protocol, including security properties. In order to give an idea of a complete characterization, we show a complete characterization of the two roles of the Needham-Schroeder protocol. (In the graphical user interface this can be reconstructed by using the "Characterize roles" function found under "Verify".)

For the initiator role of the Needham-Schroeder protocol, there is only one trace pattern, shown in Figure 1 on the next page. Thus, all traces that include the initiator role, must also include the structure in the graph, which exactly corresponds to a valid protocol execution. Thus, any synchronisation claim at the end of the initiator role is correct. For the responder role, there are exactly two explicit trace patterns,

```
// PKI infrastructure
const pk: Function;
secret sk: Function;
inversekeys (pk,sk);

// The protocol description
protocol ns3(I,R) {
    role I {
        const ni: Nonce;
        var nr: Nonce;

        send_1(I,R, { ni,I }pk(R) );
        read_2(R,I, { ni,nr }pk(I) );
        send_3(I,R, { nr }pk(R) );

        claim_I1(I,Secret,ni);
        claim_I2(I,Secret,nr);
        claim_I3(I,Nisynch);
    }
    role R {
        var ni: Nonce;
        const nr: Nonce;

        read_1(I,R, { ni,I }pk(R) );
        send_2(R,I, { ni,nr }pk(I) );
        read_3(I,R, { nr }pk(R) );

        claim_R1(R,Secret,ni);
        claim_R2(R,Secret,nr);
        claim_R3(R,Nisynch);
    }
}
const Eve: Agent;
untrusted Eve;
compromised sk(Eve);
```




[Id 1] Protocol ns3, role I, claim type Reachable

Figure 1: Left: Needham-Schroeder input file. Right: Complete characterization of role I.

shown in Figure 2. The first of these corresponds to the expected protocol execution, whilst the second is exactly the man-in-the-middle attack originally found by Lowe.

All three figures are direct output of the Scyther tool. This characterization effectively shows that every attack on the authentication properties of Needham-Schroeder includes the man-in-the middle attack, as the two patterns represent a complete characterization of the responder role. Each execution history of the protocol that includes an instance of the responder role, either contains also a partner to synchronize with, or it contains the man-in-the-middle attack.

# B Technical problems with the Athena theory

There are some technical problems in the Athena theory that cast doubt over the claim of unbounded verification proofs, and these remain unsolved in the papers. In particular, our algorithm differs significantly from the algorithm sketched in [24]. In particular, there seems to be an important problem which is most obvious in Definition 4.6 and 4.7 of the journal publication [24] (note that the same problem exists in [22], and implicitly in [23]). According to Definition 4.6 of that paper, unification is defined as *interm unification*, roughly corresponding to the notion that it is possible to interm-unify a term $t1$ not only with $t1$ but also

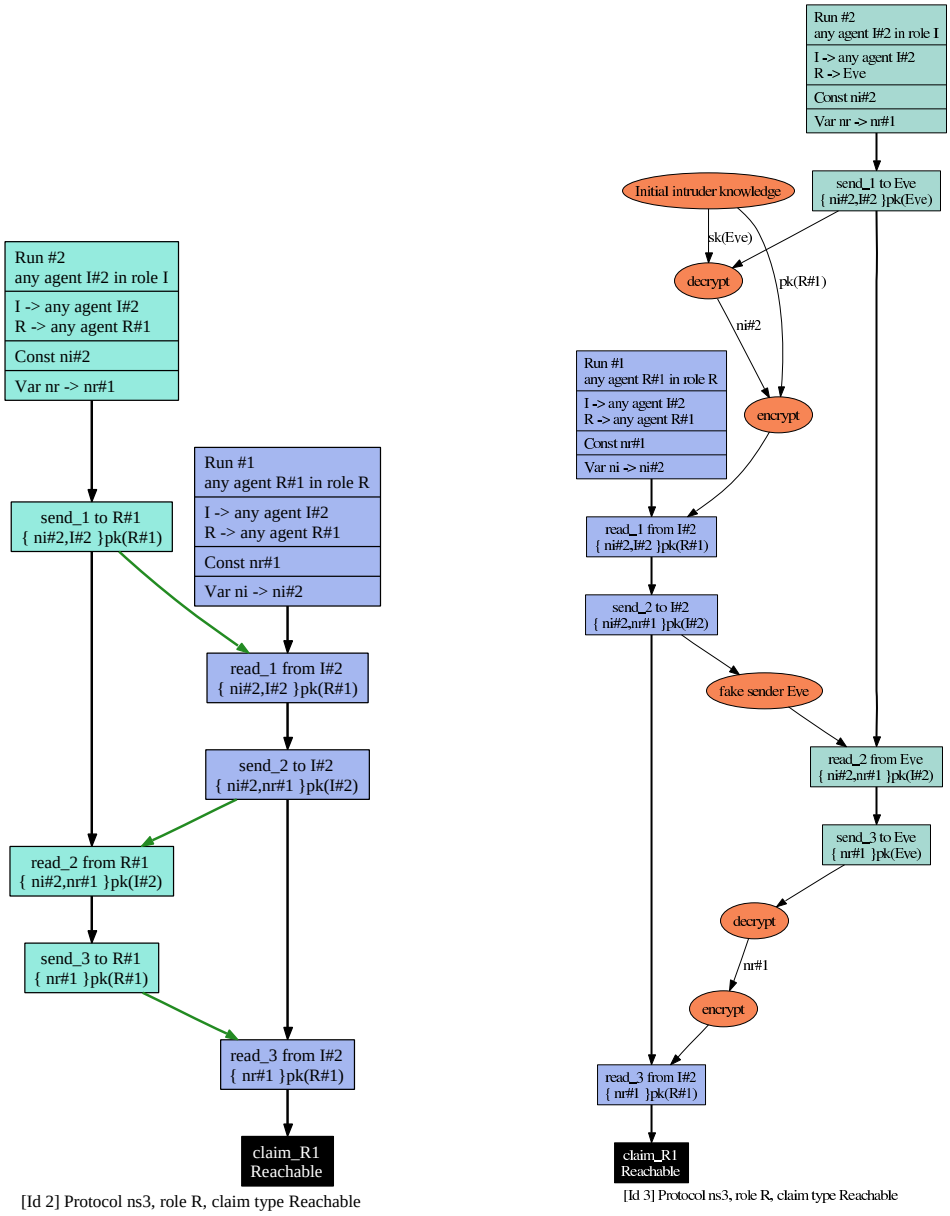



Figure 2: Needham-Schroeder, role R complete characterization: exactly two patterns

with a tuple $(t1, t2)$. However, for this relation there exists no most general interm unifier if variables are allowed to be instantiated with tuples. In fact, there exists no most general interm unifier, but rather an infinite set of incomparable unifiers for this relation.

The problem of instantiating variables with tuples can be avoided in our version of the algorithm, as we do not model intruder events as protocol actions. The additional events *decr* and *encr* are only constructed

in Scyther on-demand by the refinement process, and thus we can restrict ourselves to variables that are not instantiated as tuples.

As an example, consider the interm unification of a term $A$ with a variable $V$. We have that $A$ interm-unifies with $A$, as well as with $(V', A)$, but also with $((V', A), V'')$, etc.. As the Athena algorithm is based on the Strand Spaces approach, which models the intruder actions as protocol events, it is strictly required that variables can be instantiated with tuples, in order to correctly model encryption and decryption as protocol events. Consequently, the set $U_P(t)$ defined below Definition 4.7 in the same paper cannot be guaranteed to be finite. As a result, the next state function $\mathcal{F}$ is not complete-inclusive, which is required for the completeness of the method. The upshot of this is that attacks might be missed by the described algorithm. One of the authors of the Athena paper seems to be aware of this problem, as described in a technical report [3], where a possible fix is suggested by using so-called *interm constraints*. However, the suggested solution is described as being "possibly undecidable".