# A Syntactic Criterion for Injectivity of Authentication Protocols

C.J.F. Cremers [a] S. Mauw [a] E.P. de Vink [a,b]

[a] *Eindhoven University of Technology, Department of Mathematics and Computer Science, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands*

[b] *LIACS, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, the Netherlands*

**Abstract**

Injectivity is essential when studying the correctness of authentication protocols, because non-injective protocols may suffer from replay attacks. The standard ways of verifying injectivity either make use of a counting argument, which only seems to be applicable in a verification methodology based on model-checking, or draw conclusions on the basis of the details of the data-model used. We propose and study a property, the *loop* property, that can be syntactically verified and is sufficient to guarantee injectivity. Our result is generic in the sense that it holds for a wide range of security protocol models, and does not depend on the details of message contents or nonce freshness.

*Key words:* Security Protocols, Injectivity, Authentication, Synchronization

## 1 Introduction

The security property studied the most in the field of security protocol analysis is *authentication*. Contrary to the requirement of *secrecy*, there is no general consensus on the meaning of authentication. In fact, as indicated by Lowe [10], there is a hierarchy of authentication properties, the most popular of which is *agreement*. Agreement means that two parties involved in a protocol are guaranteed to agree upon the values of variables after successful completion of the protocol.

In [4] we extended Lowe's hierarchy by introducing the notion of *synchronization*, a security property requiring that all protocol messages occur in the expected order with the values as expected. Thus, synchronization is an intensional property, as defined by Roscoe in [13]. It can be easily shown that the synchronization property implies agreement. Moreover, assuming the standard Dolev-Yao intruder model, it is strictly stronger than agreement; some protocols satisfy agreement but not synchronization. The differences are

*This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science*
*URL:* www.elsevier.nl/locate/entcs

rather subtle, which is evidenced by the fact that almost all familiar (correct) security protocols satisfy the synchronization requirement.

It is well known that security properties satisfying agreement may still be vulnerable to so-called *replay attacks*. In a replay attack the intruder replays a message taken from a different context, thereby fooling the honest participants into thinking they have successfully completed the protocol run [11]. The left-hand protocol in Figure 1 shows an example of a protocol where the parties agree upon the values of the variables (i.e. nonce $nR$), while the right-hand scenario shows a replay attack on this protocol. The drawings are in the form of a Message Sequence Chart and must be interpreted as follows. The responder $R$ has a public/private key pair $pkR, skR$, creates a nonce, and sends his encrypted identification message to the inititiator. After reception of this message, the initiator $I$ can conclude that he shares the value of $nR$ with the responder, as expressed in the hexagon. In the right-hand MSC, two agents $a$ and $b$ execute this protocol in the roles of initiator and responder, respectively. Now, the intruder can overhear the message sent and can fool $a$ in a future run to think that $b$ has sent this message.

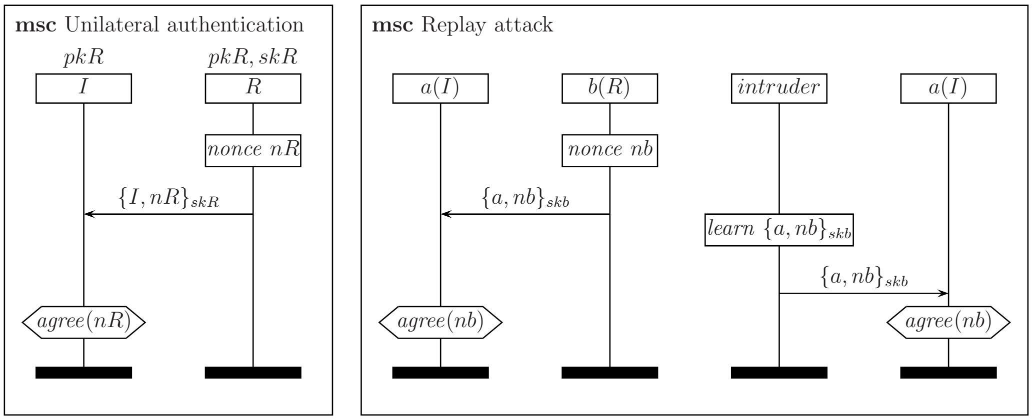


Fig. 1. An authentication protocol that is vulnerable to a replay attack.

In order to rule out such flawed protocols, the additional property of *injectivity*, proposed in [10], is required. This amounts to requiring that each run of an agent executing the initiator role corresponds to a *unique* run of its communication partner running the responder role. The unilateral authentication protocol from Figure 1 clearly does not satisfy injectivity, as is shown by the replay attack in the right-hand side of the figure. A simple fix would be to have the initiator determine the value of the nonce, as in Figure 2.

The introduction of a causal chain of messages from the initiator to the responder and back to the initiator seems to do the trick. We will call such a chain a *loop*. This property plays a key role in the discussion on injectivity below.

It is folklore that e.g. a nonce handshake is sufficient to ensure injectivity. Here we identify a more abstract property, viz. the occurrence of a loop, which

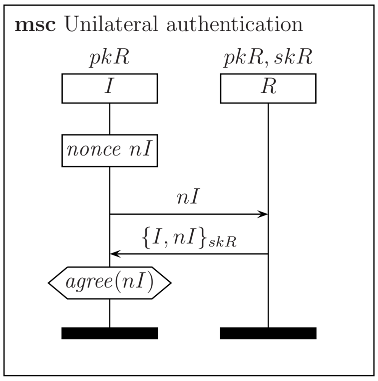


Fig. 2. Fixing the injectivity problem.

is independent of the data model, and thus applicable to a wide range of security protocol models. To give an indication of the limitations of the data based approach, consider a protocol where a nonce $n$ is created, and some function is applied to it. The result $f(n)$ is sent to the responder, who applies another function and replies with $g(f(n))$. Now, to check whether such a protocol can be injective based on the freshness of $n$ in a data-based model, we need to know some details of $f$ and $g$. If for example $f(x) = x \mod 2$, the protocol will not be injective. Our method does not require any information about the contents of messages.

Most approaches to injectivity make Lowe's definition of injectivity more precise by using a *counting* strategy: in any possible execution of the protocol the number of initiator runs may not exceed the number of corresponding responder runs. This counting argument can easily be used in a model-checking approach. Indeed, this is how injectivity is verified in the Casper/FDR tool chain [9,14]. Since it is only possible to model a finite and fixed number of scenarios, this approach will only provide an approximation of injectivity. Other approaches to the verification of security protocols, e.g. those based on logics (such as [2]) or on term rewriting (such as [5]) do not seem to pay much attention to injectivity. The Strand Spaces [15] approach, does not provide formal means to deal with injectivity. Instead, it is proposed to check authentication based on nonces, for example by using so-called *solicited authentication tests* as defined in [6]. These tests guarantee injectivity, based on nonce freshness. Authentication and injectivity are strongly connected in this view.

We mention two examples of security protocol formalisms that deal explicitly with injectivity. Gordon and Jeffrey have devised a method [7] to verify injective correspondence relations for the $\pi$-calculus. This method allows for verification by type-checking (injective) correspondences. A second example is the electronic commerce protocol logic by Adi, Debbabi and Mejri [1], where pattern matching is used to express injectivity for two-party protocols. However, it is not clear how this can be verified efficiently.

In this paper we study the question whether there is a method to validate injectivity of a security protocol which is generic in the sense that it can be applied within a large class of verification methodologies, regardless of the data model that is used. Starting point of our research is the definition of injective synchronization as provided in [4]. This definition of injectivity does not exploit counting, but simply amounts to requiring that the function which assigns agents to protocol roles is an injective function. We will review the details of this definition later. Our main result is that, for a large class of security protocol semantics, the loop property introduced above guarantees that a synchronizing protocol is also injective.

The class of security protocol semantics for which our result holds, is characterized by the closure of the set of execution traces under swapping of events. This class contains e.g. the process algebraic approach with the standard Dolev-Yao intruder model. Apart from this swap property, we will need no other assumptions on the data model and the intruder model. Since the loop property can easily be verified by means of static analysis of the security protocol description, we provide a practical syntactic criterion for verifying injectivity.

The achievement of this paper is that we have identified a general property that allows for a modular proof of injective synchronization: once non-injective synchronization has been proven, it is easy to prove injective synchronization. Interestingly, the property does not depend on the data model, and therefore does not rely on the properties of e.g. nonces, or functions that are applied to the nonces.

The remainder of this paper is structured as follows. In Section 2 we describe a formal model of security protocols and the underlying assumptions of our main result. In Section 3 we formalize injective authentication and the loop property and we prove their relation. Finally, in Section 4 we draw some conclusions and indicate options for future research.

## 2 Security protocols

A security protocol is abstractly defined as a mapping from some finite set *Role* of roles to the set $\mathit{RoleEvent}^*$ of finite sequences of so-called role events taken from the set *RoleEvent*. For a security protocol $p\colon \mathit{Role} \to \mathit{RoleEvent}^*$, a role $r \in \mathit{Role}$ represents a principal taking part in $p$ with activity $p(r)$. Sending and receiving messages is amongst such activity, as well as making claims. We assume

$$\mathit{RoleEvent} \supseteq \{\, \mathit{send}_\ell(r, r', m), \mathit{read}_\ell(r, r', m), \mathit{claim}_\ell(r, c) \mid \\ \ell \in \mathit{Label}, r, r' \in \mathit{Role}, m \in \mathit{RoleMess}, c \in \mathit{Claim} \,\}$$

The role event $\mathit{send}_\ell(r, r', m)$ is interpreted as: role $r$ sends message $m$ to be delivered to role $r'$, and $\mathit{read}_\ell(r, r', m)$ means that role $r'$ reads a message of the form $m$, with sender $r$. For an event $\mathit{claim}_\ell(r, c)$, we have that whenever $r$ claims $c$, claim $c$ is true.

Role events are all decorated with labels, typically $\ell$ taken from the set *Label*. Each role event in a security protocol has a unique label, except for corresponding send and read events. We say that the send event $send_\ell(r, r', m_1)$ and $read_\ell(r, r', m_2)$ match with sender $r$, receiver $r'$. (Note that we allow for the messages being different as, generally, the perspective of the sender and that of the receiver differ too.) The format of messages in *RoleMess* is left unspecified, as is the format of claims in *Claim*. Other types of role events may be present in *RoleEvent* as well.

The causality relation $\preccurlyeq_p$ defines a partial order on the events in a security protocol $p$. It is defined as the least reflexive and transitive relation on *RoleEvent* such that

- $e \preccurlyeq_p e'$ if $e$ precedes $e'$ in $p(r)$ for some role $r$;
- $e \preccurlyeq_p e'$ if $e = send_\ell(r, r', m) \in p(r)$, $e' = read_\ell(r, r', m) \in p(r')$ for some roles $r, r'$.

*Example* The well-known Needham-Schroeder-Lowe protocol (NSL) [12,8] as depicted in Figure 3 on the left, has two roles: that of an initiator $I$ and of a responder $R$. The injective synchronization claims at the end of the roles, here abbreviated to *i-synch*, will be defined in Section 3.1.

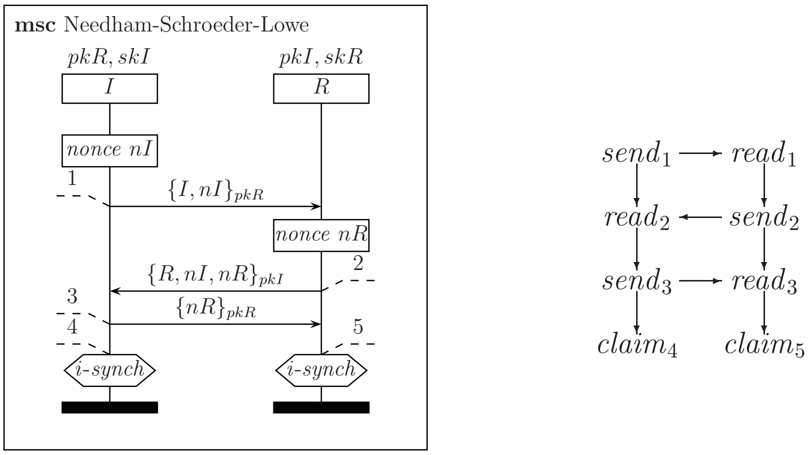


Fig. 3. The NSL protocol and the partial ordering on its events.

It is assumed that both agents know the public key of the other and the private key of their own. Formally we have

$$
\begin{aligned}
NSL(I) = {} & send_1(I, R, \{I, nI\}_{pkR}) \cdot read_2(R, I, \{R, nI, nR\}_{pkI}) \cdot \\
& send_3(I, R, \{nR\}_{pkR}) \cdot claim_4(I, i\text{-}synch) \\
NSL(R) = {} & read_1(I, R, \{I, nI\}_{pkR}) \cdot send_2(R, I, \{R, nI, nR\}_{pkI}) \cdot \\
& read_3(I, R, \{nR\}_{pkR}) \cdot claim_5(R, i\text{-}synch).
\end{aligned}
$$

The causality preorder $\preccurlyeq_p$ for NSL (of which $send_1$ is the smallest element) is given by the lattice on the right in Figure 3.

The semantics of a security protocol is a set of traces. However, in order to deal with the possibility of several role definitions being executed by the same party, we first introduce some additional machinery. Let *Agent* be the set of agents. A role executed by an agent is called a run. Because each agent can execute multiple (possibly identical) instances of a role, we assume that a unique identifier *rid* from *RunId* has been assigned to each run. An event $e$ with run identifier $rid$ is denoted by $e\sharp rid$. This yields a set *RunEvent* of so-called run events:

$$\begin{aligned} \mathit{RunEvent} \supseteq \{\ & \mathit{send}_\ell(a,a',m)\sharp rid, \mathit{read}_\ell(a,a',m)\sharp rid, \mathit{claim}_\ell(a,c)\sharp rid \mid \\ & rid \in \mathit{RunId}, \ell \in \mathit{Label}, a, a' \in \mathit{Agent}, m \in \mathit{RunMess}, c \in \mathit{Claim}\ \} \end{aligned}$$

The interpretation of, e.g., $\mathit{send}_\ell(a,a',m)\sharp rid$ is that agent $a$ sends during execution of its run $rid$ a message $m$ to agent $a'$.

The precise semantics of a security protocol is left implicit in this paper. For our purposes it suffices to assume that the semantics captures the operational behaviour of agents and some particular intruder model and that the semantics yields a collection of traces. Moreover, the semantics takes care, by means of run identifiers, of multiple instances of the security protocol. The semantics of a security protocol $p$ is denoted by $\mathit{Tr}(p) \subseteq \mathit{RunEvent}^*$. A trace $\alpha \in \mathit{Tr}(p)$ is an interleaving of runs. For a such trace $\alpha$, we use $\alpha_i$ to denote the $i$-th event of $\alpha$.

Please note that the labels and run identifiers are not part of the messages, and are therefore not under control of the intruder. They represent information that is often left implicit in the semantical models. The label indicates the state of the agent when the event is executed, and the run identifier expresses in which run an event takes place.

The reader may fill in his or her favourite semantics for the mapping *Tr*. However, we need two general requirements on this mapping for the reasoning below. We assume the semantics $\mathit{Tr}(p)$ to have the following two properties:

– The trace set $\mathit{Tr}(p)$ is closed with respect to non-read swaps, i.e., for all events $e' \neq \mathit{read}_\_(\_,\_,\_)$ it holds that

$$\alpha; e\sharp rid; e'\sharp rid'; \alpha' \in \mathit{Tr}(p) \ \Rightarrow\ \alpha; e'\sharp rid'; e\sharp rid; \alpha' \in \mathit{Tr}(p)$$

for all traces $\alpha, \alpha'$, events $e$ and two run identifiers $rid \neq rid'$.

– The trace set $\mathit{Tr}(p)$ is closed with respect to read swaps, i.e.

$$\begin{aligned} &\alpha; \mathit{send}_\ell(m)\sharp rid''; \alpha'; e\sharp rid; \mathit{read}_{\ell'}(m)\sharp rid'; \alpha'' \in \mathit{Tr}(p) \ \Rightarrow \\ &\qquad \alpha; \mathit{send}_\ell(m)\sharp rid''; \alpha'; \mathit{read}_{\ell'}(m)\sharp rid'; e\sharp rid; \alpha'' \in \mathit{Tr}(p) \end{aligned}$$

for all traces $\alpha, \alpha'$, events $e$ and run identifiers $rid, rid', rid''$ such that $rid \neq$

$rid'$.

These properties state that we can shift a non-read event to the left as long as it doesn't cross any other events of the same run. For the read event we have an additional constraint: we can only shift it to the left if there remains an earlier send of the same message.

## 3 A Syntactic Criterion for Injectivity

In this section we define a syntactic criterion for authentication protocols, called the loop property, and show that it suffices to prove injectivity. First we define a strong authentication property, called synchronization. Second, we define the syntactic criterion and prove the main result.

### *3.1 Synchronization*

Synchronization is a strong form of authentication. Whenever an agent claims synchronization, we require the interaction thus far being well behaved: all events that lead up to the claim must have occurred exactly as expected. See [4].

The general definition of synchronization refers to a specific protocol $p$ and a claim label. For simplicity of notation, we will assume in the following that there is a single protocol $p$ having a single claim role event denoted by *claim*. However, note that multiple claim run events may occur in a trace. This is the case when the role carrying the claim has been assigned to various agents or more than once to the same agent.

Consider the protocol in Figure 2, and assume the *agree* claim has been replaced by a *ni-synch* claim. The protocol claim is valid iff for all traces of the protocol, each instantiated claim in the trace is valid. Thus, if in a trace a run with identifier $rid1$ claims synchronization, we don't know who the communication partner is, but we expect that there is another run $rid2$ that executes the other role $R$, such that the following events occur in the trace in the given order:

(i) Run $rid1$ sends a nonce.

(ii) Run $rid2$ reads this exact nonce.

(iii) Run $rid2$ sends a message containing the signed nonce.

(iv) Run $rid1$ reads this exact message.

(v) Run $rid1$ claims synchronization.

As the run $rid2$ fulfills the role $R$ in this example, we require that such a run exists for each *ni-synch* claim instance. Thus, given a trace, we require for each claim instance that there are runs, that fulfill the other roles. This relation is given by a role instantiation function *cast*, with type $\{\, rid \mid \exists_i \alpha_i = claim\sharp rid \,\} \times Role \to RunId$. For each claim in the trace, it relates runs to each role, i.e. the function gives for each claiming run a complete *cast* for the protocol. In the above example we would have: $cast(rid1, I) = rid1$ and $cast(rid1, R) = rid2$.

This *cast* is not fixed. We require that for each claim run, for each run of the role of the protocol in which the claim occurs, that there exist runs that fulfill the other roles. The *cast* function is the result of combining, for all claims in the trace, the role instantiations for that claim.

In order to define synchronization formally, we introduce the functions *sendrole* and *readrole* to determine for a given label the sending role and the receiving role, respectively.

$$\begin{aligned} sendrole(\ell) = r \text{ if } send_\ell(r, r', m) \in p(r) \text{ or } read_\ell(r, r', m) \in p(r') \\ readrole(\ell) = r' \text{ if } send_\ell(r, r', m) \in p(r) \text{ or } read_\ell(r, r', m) \in p(r') \end{aligned}$$

Furthermore, we use *claimrole* to denote the name of the role in which the claim of the protocol occurs. Note that in a trace there can be more than one instance of this role.

Next, we express when a claim instance in a trace is valid. Given a trace $\alpha$, a claiming run *rid*, and a relation *cast* that maps the roles to runs, we express the auxiliary predicate $\chi$ on the domain $Tr(p) \times (RunId \times Role \to RunId) \times RunId$ by

$$\begin{aligned} \chi(\alpha, cast, rid) \iff\ & cast(rid, claimrole) = rid \wedge \\ & \forall_{read_\ell(\_) \preccurlyeq_p claim}\ \exists_{i,j \in N, a,b \in Agent, m \in RunMess} \\ & \quad i < j \\ & \quad \wedge\ \alpha_i = send_\ell(a, b, m)\sharp cast(rid, sendrole(\ell)) \\ & \quad \wedge\ \alpha_j = read_\ell(a, b, m)\sharp cast(rid, readrole(\ell)) \end{aligned} \tag{1}$$

The first conjunct of this predicate expresses the fact that the run executing the claim role is fixed by the parameter *rid*. The second conjunct expresses that in the trace $\alpha$, the claim of run *rid* is valid with respect to the specific *cast*, i.e. that the partners have executed all communications as expected. In the formula this is expressed by the fact that send and read events are executed by the expected runs, with identical message $m$, and in the right order $i < j$.

Using this predicate, we define non-injective synchronization as the validity of all claims that occur in the trace, for some *cast* function:

$$\textit{NI-SYNCH} \iff \forall_{\alpha \in Tr(p)} \exists_{cast}\ \forall_{i,rid}\ \alpha_i = claim\sharp rid \Rightarrow \chi(\alpha, cast, rid)$$

Thus, if for all traces a *cast* exists, such that for every run that claims synchronisation the $\chi$ predicate holds, the protocol synchronizes. Provided that $\chi(\alpha, cast, rid)$ holds, *cast* has the following two properties: First, each claim run fulfills the claim role itself, and thus: $\forall_{rid}\ cast(rid, claimrole) = rid$. Second, *cast* only assigns runs to roles, when these runs are executing that role, so we have: $\forall_{rid,r}\ role(cast(rid, r)) = r$.

The cast function provides a natural handle for introducing injectivity. Injectivity requires that two different claim instances synchronize with different partner runs. Thus, we say that a role instantiation function *cast* is injective if

$$\forall_{(rid1,r1)\neq(rid2,r2)}\ cast(rid1, r1) \neq cast(rid2, r2)$$

We incorporate this injectivity requirement into the definition of *NI-SYNCH*. The definition of *injective* synchronization is then given by

$$I\text{-}SYNCH \iff \forall_{\alpha\in Tr(p)}\exists_{cast \text{ injective}}\ \forall_{i,rid}\ \alpha_i = claim\sharp rid \Rightarrow \chi(\alpha, cast, rid)$$

So, on top of the synchronization requirement, we now demand that for all communications preceding the claim of the protocol, there are unique runs executing the protocol roles.

Having defined synchronization formally, we investigate some properties of synchronizing protocols. Given a valid synchronization claim of a run *rid* in a trace $\alpha$, there exists a role instantiation function *cast* such that $\chi(\alpha, cast, rid)$ holds. The predicate $\chi$ tells us that certain events exist in the trace. Because we want to reason about these events in the following, we decide to make this set of events explicit. Slightly abusing notation, we use $\overline{e}$ to denote the unique role event corresponding to the run event $e$. The function $runid(e)$ simply yields the run identifier of a run event $e$. We define the set of events $\chi'(\alpha, cast, rid)$ by

$$\chi'(\alpha, cast, rid) = \{\ e \mid \exists_i\ \alpha_i = e \wedge runid(e) = cast(rid, role(e)) \wedge \overline{e} \preccurlyeq_p claim\ \}$$

Assuming that $\chi$ holds, its set of events $\chi'$ has two interesting properties. If there is a read in this set, there is also a matching send in the set. Furthermore, given an event of a role in the set, all preceding events of the same role are also in the set.

To prove our main result, the two swap properties introduced in Section 2 suffice. However, to ease the explanation of the proof, we introduce two additional lemmas. These lemmas are implied by the model and the two swap properties.

The first lemma generalizes the swapping of two events to the swapping of a set of events. The lemma does not hold for any set of events: we now use results obtained for a set of events defined by $\chi$, that are involved in a synchronisation claim. Based on the two swap properties, we can shift these events (in their original order) to the beginning of the trace. To express such shifting of sets of events, we introduce a shift function on traces. We define a trace transformation function $shift : \mathcal{P}(RunEvent) \times RunEvent^* \to RunEvent^*$

by

$$shift(E, \alpha) = \begin{cases} \alpha & \text{if } \forall_i(\alpha_i \notin E) \\ e; shift(E, \beta; \beta') & \text{if } \alpha = \beta; e; \beta' \wedge \forall_i(\beta_i \notin E) \wedge e \in E \end{cases}$$

This function effectively reorders a trace. The next lemma formulates conditions assuring that the reordering of a trace in $Tr(p)$ is in $Tr(p)$ as well.

**Lemma 3.1** *Given a protocol $p$ and a trace $\alpha \in Tr(p)$, claim run rid and role instantiation function cast:*

$$\chi(\alpha, cast, rid) \;\wedge\; \alpha' = shift(\chi'(\alpha, cast, rid), \alpha) \;\Rightarrow \\ \alpha' \in Tr(p) \;\wedge\; \chi(\alpha', cast, rid)$$

**Proof.** Induction on the size of the finite set $\chi'(\alpha, cast, rid)$, because $\chi(\alpha, cast, rid)$ implies that the read events can be swapped. □

The lemma directly generalizes to more claim instances (of the same claim). Thus, instead of a single claim run, we can consider sets of claim runs.

**Lemma 3.2** *Given a trace $\alpha \in Tr(p)$, a set of claim runs $CR \in \mathcal{P}(RunId)$ and instantiation function $cast : RunId \times Role \rightarrow RunId$:*

$$(\forall_{rid \in CR}\ \chi(\alpha, cast, rid)) \;\wedge\; \alpha' = shift(\bigcup_{rid \in CR} \chi'(\alpha, cast, rid), \alpha) \;\Rightarrow \\ \alpha' \in Tr(p) \;\wedge\; (\forall_{rid \in CR}\ \chi(\alpha', cast, rid))$$

**Proof.** Similar to the proof of Lemma 3.1. □

If we apply the *shift* function to a trace of the system, and the conditions of the lemma are met, we get a reordered trace, that is also in $Tr(p)$. The new trace consists of two segments: in the first segment there are only the preceding events of the claim runs in $CR$, and all other events are in the second segment.

Intuitively, these lemmas express that the events involved in a valid synchronization claim are independent of the other events in the trace. A valid synchronization can occur at any point in the trace, because it does not require the involvement of other runs, or of the intruder. However, other events in the trace might depend on events involved in the synchronization. Thus we cannot shift the synchronizing events to the right; but we can shift them to the left, which ensures that any dependencies are not be broken.

We use these lemmas in the injectivity proof in the next section.

### *3.2 The LOOP property*

We define a property of protocols, which we call the *LOOP* property. For protocols with only two roles, it resembles a ping-pong property: First the claim role executes an event, then the other role, and then the claim role again. For example, the *LOOP* property does not hold for the protocol in Figure 1, but it does hold for the protocols in Figures 2 and 3.

We generalize this for multiparty protocols with any number of roles. We require that the partner roles have an event that must occur after the start of the claim run, but before the claim event itself.

**Definition 3.3** A security protocol $p$ has the *LOOP* property iff

$$\forall_{e \preccurlyeq_p claim, role(e) \neq claimrole} \exists_{e', e''} \\ e' \preccurlyeq_p e'' \preccurlyeq_p claim \ \wedge \ role(e') = claimrole \ \wedge \ role(e'') = role(e) \quad (2)$$

Now we can state the main theorem, which gives us a syntactic condition for the injectivity of a protocol that synchronizes.

**Theorem 3.4**

$$NI\text{-}SYNCH \wedge LOOP \Rightarrow I\text{-}SYNCH$$

**Proof.** By contradiction. Assume that the implication does not hold. Thus we have

$$NI\text{-}SYNCH \wedge LOOP \wedge \neg I\text{-}SYNCH \quad (3)$$

The remainder of the proof is done in two steps. The first step of the proof establishes a trace $\alpha$ of the protocol, in which there are two runs that synchronize with the same run. In the second step we use the shifting lemmas to transform $\alpha$ into another trace of the protocol. For this new trace, we will show that *NI-SYNCH* cannot hold, which contradicts the assumptions.

From now on, we will omit the type information for $\alpha$ and *cast* in the quantifiers and assume that $\alpha \in Tr(p)$. Given that the protocol synchronizes, but that it is not injective, we derive from the definitions of *NI-SYNCH*, *I-SYNCH* and formula (3) that

$$\forall_\alpha \exists_{cast} \forall_{i,rid}\, \alpha_i = claim\sharp rid \Rightarrow \chi(\alpha, cast, rid) \wedge \\ \neg \forall_\alpha \exists_{cast \text{ injective}} \forall_{i,rid}\, \alpha_i = claim\sharp rid \Rightarrow \chi(\alpha, cast, rid) \quad (4)$$

We push the negation on the right through the quantifiers, yielding

$$\forall_\alpha \exists_{cast} \forall_{i,rid}\, \alpha_i = claim\sharp rid \Rightarrow \chi(\alpha, cast, rid) \wedge \\ \exists_\alpha \forall_{cast} \neg(cast \text{ injective} \wedge \forall_{i,rid}\, \alpha_i = claim\sharp rid \Rightarrow \chi(\alpha, cast, rid)) \quad (5)$$

Based on the existential quantifiers in (5), we choose a trace $\alpha$ and instantiation function *cast* such that

$$\forall_{i,rid}\, \alpha_i = claim\sharp rid \Rightarrow \chi(\alpha, cast, rid) \wedge \\ \neg(cast \text{ injective} \wedge \forall_{i,rid}\, \alpha_i = claim\sharp rid \Rightarrow \chi(\alpha, cast, rid)) \quad (6)$$

Note that in (6) the left conjunct also occurs as a sub-formula in the right conjunct. Rewriting yields

$$\forall_{i,rid}\ \alpha_i = \mathit{claim}\sharp rid \Rightarrow \chi(\alpha, \mathit{cast}, rid)\ \wedge\ \neg \mathit{cast}\ \text{injective} \tag{7}$$

Making the non-injectiveness for the function *cast* explicit, there must exist two run identifiers, for which $\chi$ holds:

$$\exists_{(rid1,r1),(rid2,r2),rid3}\ \chi(\alpha, \mathit{cast}, rid1)\ \wedge\ \chi(\alpha, \mathit{cast}, rid2)\ \wedge \\ \mathit{cast}(rid1, r1) = \mathit{cast}(rid2, r2)\ \wedge\ (rid1, r1) \neq (rid2, r2) \tag{8}$$

From the predicate $\chi$ and formula (8), we have that the run $\mathit{cast}(rid1, r1)$ must be executing the role $r1$. Because $\mathit{cast}(rid1, r1) = \mathit{cast}(rid2, r2)$ it is also executing role $r2$. As runs only execute a single role, it must be that $r1 = r2$. The inequality from the existential quantifier now reduces to $rid1 \neq rid2$.

Put $r = r1 = r2$. We choose two run identifiers $rid1, rid2$ such that Formula (8) holds for $r$. Now there exists a run identifier $rid3$ such that

$$\mathit{cast}(rid1, r) = \mathit{cast}(rid2, r) = rid3$$

From the definition of $\chi$, we obtain that if $r$ would be equal to *claimrole*, we would have $rid3 = rid1$ and $rid3 = rid2$, implying $rid1 = rid2$ and contradicting Equation (8). Thus, we have $r \neq \mathit{claimrole}$.

We have now established that the trace $\alpha$ contains at least three runs. Two of these, $rid1$ and $rid2$, are executing the claim role, while the third, $rid3$ is executing a different role $r$. Furthermore, we have that the claims of $rid1$ and $rid2$ synchronize with $rid3$. This completes the first step of the proof. We will now proceed by transforming $\alpha$ into a trace for which *NI-SYNCH* cannot hold, for the second part of the proof.

Because we have $\chi(\alpha, \mathit{cast}, rid1)$ and $\chi(\alpha, \mathit{cast}, rid2)$, on the basis of Lemma 3.2 we can apply *shift* using $rid1$ and $rid2$ to get a trace $\alpha' \in \mathit{Tr}(p)$

$$\alpha' = \mathit{shift}(\chi'(\alpha, \mathit{cast}, rid1) \cup \chi'(\alpha, \mathit{cast}, rid2), \alpha)$$

In the trace $\alpha'$ we now have two distinct segments. All events involved with the synchronization of $rid1$ and $rid2$ are now in the initial segment of $\alpha'$. This includes the events of $rid3$ that precede the claim. The second segment of $\alpha'$ contains all other events, that are not involved in the preceding events of $rid1$ and $rid2$.

We will now reorder the initial segment of $\alpha'$. To this end, we apply the *shift* function a second time, now only for $rid1$. This will also yield a trace of the protocol, because the conditions of Lemma 3.2 hold for $\alpha'$, as the application of *shift* to $\alpha$ maintained the order of the events in the shifted set, which implies that $\chi(\alpha', \mathit{cast}, rid1)$ holds. Thus, we also know that the following trace is an element of $Tr(p)$:

$$\alpha'' = shift(\chi'(\alpha', cast, \{rid1\}), \alpha')$$

Because the *shift* function maintains the order of the involved events, we have that $\alpha'' = \beta; \gamma; \delta$, where

$$\forall_i \; \beta_i \in \chi'(\alpha', cast, \{rid1\})$$
$$\forall_i \; \gamma_i \in \chi'(\alpha, cast, \{rid1, rid2\}) \setminus \chi'(\alpha', cast, \{rid1\})$$

All events that are not involved with the synchronization claims of $rid1$ and $rid2$, are in $\delta$.

Observe that $\beta$ includes all events of $rid3$ that are involved with the claim of the run $rid1$. As all events are unique, these are not part of $\gamma$. From the construction of the involved events set, we know that all involved events of role $r$ are those of run $rid3$, because all other runs are executing other roles (as indicated by *cast*). This implies that there are no events of role $r$ in the $\gamma$ at all: these are all in $\beta$.

Now we have arrived at a contradiction. $\alpha''$ is in the set $Tr(p)$. The loop property combined with *NI-SYNCH* requires that for each role, there is an event after the first event of the claim role, that occurs before the claim. For the run $rid2$ all events are in $\gamma$ (including the start and the claim), but in this segment there is no event of role $r$. Thus, there can be no *cast* for $\alpha''$ such that $\chi(\alpha'', cast, rid2)$ holds. This implies that *NI-SYNCH* does not hold for the protocol, which contradicts the assumptions. □

## 4 Conclusions and Future Work

We have shown that for a large class of security protocol models, injectivity of authentication protocols is easy to verify, once synchronization has been established. Until now, injectivity and authentication have been strongly connected. Our new results establish that it suffices to verify the non-injective variant of synchronization. Verifying injectivity is a simple and separate task, which does not depend on any specific (data) model.

We did not choose a specific security protocol model for this result. Instead, we have characterized a class of models in which the theorem holds. This class contains nearly all models found in the literature, such as the Strand Spaces model, Casper/FDR without time, and term rewrite systems [15,9,5]. These models share the following properties:

- Multiple instances of the protocol are truly independent. They do not share variables, memory, or time.
- The intruder has the ability to duplicate messages. This holds, for example, in the standard Dolev-Yao intruder model.

The question arises whether the theorem also holds in an intruderless model. This is in fact the case, but of less interest, because injectivity always holds for synchronizing or agreeing protocols when there is no intruder.

Verifying the *LOOP* property can be easily implemented. We are currently working on an extension of the Scyther tool [3]. The algorithm is an instance of the reachability problem in an acyclic finite graph, and therefore linear.

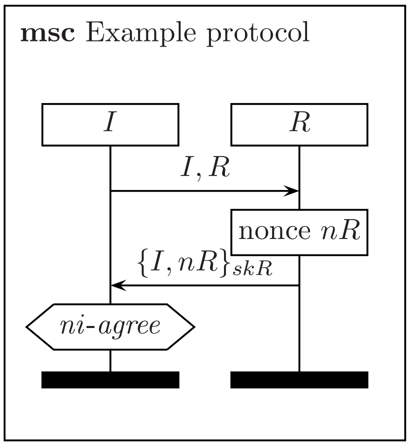


Fig. 4. A unilateral agreement protocol, with LOOP, but not injective.

Almost all correct protocols satisfy *NI-SYNCH* as well as *LOOP*. It seems that *LOOP* is a necessary condition for injectivity. We know that this holds for the Dolev-Yao intruder model. However, for peculiar intruder models *LOOP* is not a necessary condition for injectivity.

In the models where *LOOP* is also a necessary condition for injectivity, our results imply a minimum number of messages in a multi-party authentication protocol. We will investigate this in future work.

Here we have only considered synchronization, and not agreement. This raises the question whether there is a similar property to show injectivity of agreeing protocols. It can be seen from the example in Figure 4, that *LOOP* does not suffice to guarantee injectivity. The protocol satisfies the loop property for the claim role, and the protocol satisfies non-injective agreement, but not injective agreement. Finding an alternative for agreeing protocols is an interesting challenge for future research.

## References

[1] Adi, K., M. Debbabi and M. Mejri, *A new logic for electronic commerce protocols*, Theoretical Computer Science **291** (2003), pp. 223–283.

[2] Burrows, M., M. Abadi and R. Needham, *A logic of authentication*, ACM Transactions on Computer Systems **8** (1990), pp. 16–36.

[3] Cremers, C., *Scyther security protocol verification tool: documentation*, http://www.win.tue.nl/~ccremers/scyther.

[4] Cremers, C., S. Mauw and E. d. Vink, *Defining authentication in a trace model*, in: T. Dimitrakos and F. Martinelli, editors, *Proc. FAST 2003* (2003), pp. 131–145.

[5] Genet, T. and F. Klay, *Rewriting for cryptographic protocol verification*, in: *Conference on Automated Deduction* (2000), pp. 271–290.

[6] Guttman, J. D. and F. J. Thayer, *Authentication tests and the structure of bundles.*, Theor. Comput. Sci. **283** (2002), pp. 333–380.

[7] Jeffrey, A. and A. Gordon, *Typing One-to-One and One-to-Many Correspondences in Security Protocols*, in: M. Okada, B. C. Pierce, A. Scedrov, H. Tokuda and A. Yonezawa, editors, *Proc. ISSS '02* (2002), pp. 418–434.

[8] Lowe, G., *Breaking and fixing the Needham-Schroeder public-key protocol using FDR*, in: T. Margaria and B. Steffen, editors, *Proc. TACAS '96* (1996), pp. 147–166.

[9] Lowe, G., *Casper: A compiler for the analysis of security protocols*, in: *Proc. CSFW '97, Rockport* (1997), pp. 18–30.

[10] Lowe, G., *A hierarchy of authentication specifications*, in: *Proc. CSFW '97, Rockport* (1997), pp. 31–44.

[11] Malladi, S., J. Alves-Foss and R. Heckendorn, *On preventing replay attacks on security protocols*, in: *Proc. ICSM '02* (2002), pp. 77–83.

[12] Needham, R. and M. Schroeder, *Using encryption for authentication in large networks of computers*, Communications of the ACM **21** (1978), pp. 120–126.

[13] Roscoe, A. W., *Intensional Specifications of Security Protocols*, in: *Proc. CSFW '96* (1996), pp. 28–38.

[14] Ryan, P., S. Schneider, M. Goldsmith, G. Lowe and B. Roscoe, "Modelling and Analysis of Security Protocols," Addison Wesley, 2000.

[15] Thayer, F., J. Herzog and J. Guttman, *Strand spaces: Why is a security protocol correct*, in: *Proc. Symposium on Security and Privacy, Oakland* (1998), pp. 160–171.