Unbounded Verification, Falsification, and Characterization of Security Protocols by Pattern Refinement

Cas J.F. Cremers[*]
Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland
cas.cremers@inf.ethz.ch

ABSTRACT

We present a new verification algorithm for security protocols that allows for unbounded verification, falsification, and complete characterization. The algorithm provides a number of novel features, including: (1) Guaranteed termination, after which the result is either unbounded correctness, falsification, or bounded correctness. (2) Efficient generation of a finite representation of an infinite set of traces in terms of patterns, also known as a complete characterization. (3) State-of-the-art performance, which has made new types of protocol analysis feasible, such as multi-protocol analysis.

Categories and Subject Descriptors

C.2.2 [**Computer-communication Networks**]: Network Protocols—*Protocol verification*

General Terms

Security, Theory, Verification

Keywords

Security protocol analysis, unbounded verification, falsification, characterization

1. INTRODUCTION

The formal analysis of security protocols, which started from works such as [10, 17], has led to the development of (semi-)automatic methods for the analysis of abstract security protocols. The seminal work of Meadows and Lowe [21, 23] showed that it was feasible for automatic tools to find attacks on security protocols that had gone unnoticed by experts for years. In the following years many algorithms were developed, e. g. [16, 23, 28], and a significant number of tools were made publicly available [1, 4, 6, 8, 11, 21, 31] for the analysis of security protocols. The security properties that these tools can verify include confidentiality and various forms of authentication. However, even under the assumption of perfect cryptography, verification of such properties is undecidable [18]. We can roughly distinguish two classes of algorithms and corresponding tools, based on how they deal with this undecidability.

The vast majority of algorithms and tools deals with the undecidability by performing so-called *bounded verification*: instead of verifying the properties for all possible behaviours of the protocol, only a finite subset of behaviours is considered. At best, such tools can establish that a property holds within the finite subset considered. These methods usually bound either the message size, the number of freshly generated values (Nonces), the number of protocol sessions considered, or combinations of these.

In contrast, other algorithms and tools aim for *unbounded verification* [2, 3, 6–8, 19, 28], and try to establish that the security properties hold for all possible behaviours of a protocol in the presence of a Dolev-Yao style intruder. The current state-of-the-art tool for unbounded verification by overapproximation is *ProVerif* by Blanchet [6]. ProVerif has proven to be a very efficient tool: in most cases it is able to perform unbounded verification of small (e. g. Needham-Schroeder-Lowe) to medium sized protocols (e. g. TLS), or find attacks such as the man-in-the-middle attack on the Needham-Schroeder protocol, in less than a second. However, for some protocols and properties, ProVerif may either never terminate, or when terminating, it may not be able to make a statement about the property.

Closely related to the concept of unbounded verification is the recently introduced notion of *complete characterization*, found in e. g. [16]. Given a protocol role, for example the initiator or the responder role, a complete characterization is a finite set of representatives of all possible protocol behaviours that include an instance of the role. The representatives are constructed in such a way that they allow for verification of certain security properties. Hence, if these properties hold for each of these representatives, they must hold for all possible protocol behaviours. For a restricted class of security properties, analysis by characterization is equivalent to performing unbounded verification. Like the other unbounded algorithms, the algorithm sketched in [16], called CPSA, is in certain cases unable to provide a statement about the protocol properties. Another drawback of the CPSA algorithm is that it is strongly connected to the notion of authentication tests, and restricts the class of protocols and properties that can be handled.

[*]This work was partially supported by the Hasler Foundation, under ManCom project 2071.

Our contributions. We unify and solve the problems above by presenting a new algorithm for the analysis of security protocols. Our algorithm allows for unbounded verification and falsification of security properties, as well as complete characterization, and builds on ideas developed in [28]. For the vast majority of the verification problems, the algorithm is able to efficiently falsify or verify the property. Unlike ProVerif and CPSA, our algorithm is guaranteed to terminate, and in the case that the algorithm is not able to establish unbounded verification, it establishes a form of bounded verification that is similar to the guarantees provided by tools such as OFMC [4]. The algorithm uses no abstraction techniques, and therefore it can always generate counterexamples, as well as verify intricate authentication properties such as *synchronisation* [15]. To the best of our knowledge, it is the only unbounded verification algorithm that provides a meaningful statement about the security property in all cases. Also, the algorithm can efficiently compute a *complete characterization* [16] of protocol roles, for a larger class of protocols than can be handled by CPSA. An instance of the algorithm has been implemented in the *Scyther* tool which exhibits state-of-the-art performance [14] and has made new types of protocol analysis feasible, such as multi-protocol analysis [12].

The remainder of the paper is structured in the following way. In Section 2 we describe an abstract protocol model, which serves as a basis for our exposition. In Section 3 we introduce patterns, which are used to represent infinite sets of execution traces. In Section 4 we describe the algorithm. We discuss related work in Section 5. Future work is discussed in 6 and we conclude in Section 7.

2. SECURITY PROTOCOL MODEL

In this section we present a generic protocol model which will serve as a basis for the exposition in the following sections. The model is generic in the sense that it is compatible with most existing protocol formalisms that assume perfect cryptography.

The fundamental structure in the protocol model is the set of terms, used to represent the messages that are exchanged during a protocol session. We define the set of basic terms as the union of the set of global constants *Const*, the set of terms freshly generated during protocol execution *Nonce*, and the set of variables *Var*. The set of terms *Term* includes the basic terms and any term constructed by one of the two constructors: tupling written as (t, t') and encryption of a term t with a key t', written as $\{\!|\, t \,|\!\}_{t'}$. The set *ID* denotes the set of thread identifiers, which is explained below.

$$\begin{aligned} \mathit{BasicTerm} &::= \mathit{Const} \mid \mathit{Nonce}_{\mathit{ID}} \mid \mathit{Var}_{\mathit{ID}} \\ \mathit{Term} &::= \mathit{BasicTerm} \mid (\mathit{Term}, \mathit{Term}) \mid \{\!|\, \mathit{Term} \,|\!\}_{\mathit{Term}} \end{aligned}$$

Both symmetric and asymmetric cryptography are captured by the encryption constructor. For each term, we define the inverse key function $\cdot^{-1}$ from terms to terms, and write t^{-1} to denote the inverse key of a term t. For symmetric keys, we have that $t^{-1} = t$. Any term can be used as a key, including composed terms. We define an auxiliary function $\mathit{unpair} : \mathit{Term} \to \mathcal{P}(\mathit{Term})$, that is used to decompose tuples into sets of non-tuple terms. *unpair* is defined as $\mathit{unpair}(t) = \mathit{unpair}(t1) \cup \mathit{unpair}(t2)$ iff $t = (t1, t2)$, and $\{t\}$ otherwise.

Variables are by definition typed, but we explicitly allow for variables of type *Term*. This allows us to e. g. capture type flaw attacks, or handle ciphertext forwarding (tickets). For convenience, we define a type for all terms, given by the function $\mathit{type} : \mathit{Term} \to \mathcal{P}(\mathit{Term})$, where we have that $t \notin \mathit{Var} \Rightarrow \mathit{type}(t) = \{t\}$, and furthermore we have that for all t the elements of $\mathit{type}(t)$ are ground terms, i. e. they do not contain variables as subterms.

A substitution $\sigma = [u_1, \ldots, u_N / t_1, \ldots, t_N]$ is said to be *well-typed* iff $\forall i : 1 \leq i \leq N : \mathit{type}(u_i) \subseteq \mathit{type}(t_i)$. A variable v is called a *basic variable* iff $\mathit{type}(v) \subseteq \mathit{BasicTerm}$. We assume each variable has a type with infinite domain and that we can always construct a fresh term of that type; in particular, the intruder has this ability. We call a well-typed substitution σ a *unifier* of term t and term t' iff $\sigma(t) = \sigma(t')$. We call σ the *most general unifier* of two terms t, t', notation $\sigma = \mathit{MGU}(t, t')$, iff for any other unifier σ' there exists a substitution σ'', such that $\sigma' = \sigma \circ \sigma''$.

We assume protocols from the set *Protocol* define a finite set of roles. Each role is defined as a finite list of protocol events.

$$\mathit{ProtEvent} ::= \mathsf{send}_{\mathit{ID}}(\mathit{Term}) \mid \mathsf{recv}_{\mathit{ID}}(\mathit{Term})$$

In general, protocol descriptions may include other types of events (e. g. claims or control flow) but we abstract from them here.

A protocol role may be executed any number of times by each agent: each such execution is called a *thread.* (In other formalisms threads may be called strands, runs, or role instances.) A thread is identified by a unique identifier from the set of thread identifiers *ID*. This set consists of a placeholder τ used in protocol definitions, and an unbounded number of identifiers that is used in execution traces. In traces, we substitute in each event the placeholder τ from the protocol definition by an identifier that uniquely identifies the thread.

We assume a Dolev-Yao style intruder, which has full control over the network. Hence every sent message is immediately observed by the intruder, and any received message is supplied by the intruder. We assume the intruder can generate any number of constants of the type of each variable, and also possesses a finite set of terms stemming from the *compromised agents* with whom he conspires. This includes the private keys of these agents, and as a result the intruder can impersonate these agents.

In order to reason about the intruder knowledge in execution traces, it is common to either use implicit inference rules and closure over the events performed by agents (e. g. [26]), or introduce explicit events that represent the tupling and encryption performed by the intruder (e. g. [30]). Here we choose the middle ground as in [28]: we handle tupling implicitly, but handle encryption explicitly, by introducing explicit encr and decr events. The set of events that can occur during protocol executions consists of the protocol events, and the events of the intruder, consisting of init events that correspond to terms stemming from the initial intruder knowledge, events to denote encryption and decryption, and know events that indicate that a term is in the intruder knowledge.

$$\begin{aligned} \mathit{IntruderEvent} &::= \mathsf{init} \mid \mathsf{encr} \mid \mathsf{decr} \mid \mathsf{know} \\ \mathit{Event} &::= \mathit{ProtEvent} \mid \mathit{IntruderEvent}(\mathit{Term}) \end{aligned}$$

Given a protocol Q, we assume there is a definition of the set of possible execution traces, i. e. $\mathit{traces}(Q)$, for example as given by the operational semantics in [15].

We require from the protocol semantics that the set of traces is constrained by the protocol on the basis of (1) protocol order, i. e. an event can only occur in a specific thread if the previous events have happened in the same thread as indicated by the protocol, and (2) event enabling, i. e. receive events can only be executed if the expected message is available, and all other events can always be executed. Similarly, the intruder can only decrypt an encrypted term if he has previously learnt both the encrypted term and the key. In order for the substitution refinement (as defined in the next section) to be sound, we require that variables are assigned a value only once (no re-assignment of variables) by pattern matching. Note that e. g. recent versions of the Strand Spaces model [16] also satisfy these requirements.

We assume that traces are finite, and that the events in a trace are unique; if not, a form of labeling can be applied. For convenience, we regard each trace *tr* as a set of events with an associated total order, denoted as $tr = (E, \leq)$. For convenience, we often write tr_E to denote the set of events in a trace *tr*, and $tr_{\leq}$ to denote their order. Traces do not contain variables.

In order to capture the interaction between events and the intruder knowledge, we define two functions for each event: the input function *in* yields the terms that are required to be in the intruder knowledge to enable the event, and the output function *out* that yields the terms that are added to the intruder knowledge after an event. The functions *in* and *out* are both of type $\mathit{Event} \rightarrow \mathcal{P}(\mathit{Term})$, and are defined as in Table 1.

Table 1: *in* and *out* functions

e	$in(e)$	$out(e)$
$\mathsf{send}(t)$ or $\mathsf{init}(t)$	$\emptyset$	$unpair(t)$
$\mathsf{recv}(t)$ or $\mathsf{know}(t)$	$unpair(t)$	$\emptyset$
$\mathsf{encr}(\{\| t \|\}_{t'})$	$unpair((t, t'))$	$\{\{\| t \|\}_{t'}\}$
$\mathsf{decr}(\{\| t \|\}_{t'})$	$\{\{\| t \|\}_{t'}\} \cup unpair(t'^{-1})$	$unpair(t)$

The events of a trace *tr* are *enabled* iff $\forall e \in E_{tr}, t \in in(e) : \exists e' \leq_{tr} e : t \in out(e')$.

3. PATTERNS

We introduce the notion of *patterns* in order to reason about infinite sets of traces of a protocol. A pattern is a tuple $pt = (E, \rightarrow)$, where E is a set of events, and $\rightarrow$ is a relation on the events from E. A pattern forms a labeled directed acyclic graph (DAG). The relation $\rightarrow$ induces a partial order on the events, which generalizes the total order on the events occurring in traces. An edge $\rightarrow$ is either said to be unlabeled (written as $e \overset{\cdot}{\rightarrow} e'$) or labeled with a term t ($e \xrightarrow{t} e'$). Unlabeled edges are used to represent the order of the events within a single role instance, whereas labeled edges are used to capture the earliest point at which a certain message is known to the intruder. Labeled edges are also referred to as *bindings*, where the term t in the edge $e \xrightarrow{t} e'$ is said to be *bound* to e.

Given a pattern $(E, \rightarrow)$, we define $\rightarrow^*$ to be the reflexive, transitive closure of $\rightarrow$ (including both the labeled and unlabeled variants). We explicitly choose $\rightarrow^*$ to be reflexive (and thus cyclic) to simplify some of the definitions.

We say that a directed acyclic graph $(E, \rightarrow)$ is a pattern of a protocol Q if and only if:

1. Bindings denote message causality:
$$e \xrightarrow{t} e' \Rightarrow t \in out(e) \wedge t \in in(e').$$
2. Terms are bound to the earliest possible event:
$$\forall e, e', e'', t : (e \rightarrow^* e' \wedge e' \xrightarrow{t} e'' \wedge t \in out(e)) \Rightarrow e = e'.$$
3. The pattern is consistent with the protocol Q, i. e.
$$\forall e \in \mathit{ProtEvent} \cap E : \Big(\exists r \in Q, e' \in r, \sigma : \big(e = \sigma e' \wedge (\forall e'' : e'' \preccurlyeq_r e' \Rightarrow \sigma e'' \rightarrow^* e)\big)\Big),$$
where $\preccurlyeq_r$ denotes the order on the events of a role r of Q.

We can interpret a trace as a pattern, if we interpret the trace order $\leq$ as a set of unlabeled edges. Conversely, a pattern *pt* can be considered as a filter on the traces of Q, representing the set of traces of Q that exhibit the pattern. We extend the function *traces* to capture this set, by defining

$$\mathit{traces}(Q, pt) = \Big\{ tr \;\Big|\; tr \in \mathit{traces}(Q) \wedge \exists \sigma : \forall e, e', e'' : \Big((e \rightarrow^* e' \Rightarrow \sigma e \leq_{tr} \sigma e') \wedge \big((e \xrightarrow{t} e' \wedge e'' \leq_{tr} \sigma e \wedge \sigma t \in out(e'')) \Rightarrow e'' = \sigma e\big)\Big) \Big\},$$

where σ is a well-typed substitution from variables to terms.

For some patterns one can directly construct traces of the protocol that contain the pattern. We call these patterns *realizable patterns*, i. e. the realizable patterns are those patterns for which the following predicate is true:

$$\text{realizable}((E, \rightarrow)) \equiv \forall e \in E : \forall t \in in(e) \setminus \mathit{Var} : \exists e' \rightarrow^* e : t \in out(e') .$$

Note that here we use the assumption that the intruder can generate sufficiently many terms of each type, which allows us to ignore the elements of $in(e)$ that are variables.

From such realizable patterns, we can trivially generate traces of Q that exhibit the pattern, which is captured by the following lemma.

Lemma 1. *Let $pt = (E, \rightarrow)$ be a realizable pattern of Q. Then $\mathit{traces}(Q, pt)$ is non-empty, in particular for any total extension $\leq$ of $\rightarrow^*$, and any well-typed substitution σ of variables to terms that does not violate the earliest-binding requirement on patterns, we have that $\sigma(E, \leq) \in \mathit{traces}(Q, pt)$.*

For patterns that do not satisfy the realizable predicate, it is not immediately clear whether they represent any traces of the protocol. We introduce the notion of pattern refinement that allows us to manipulate patterns, with the ultimate goal of arriving at realizable patterns.

Patterns can be *refined* by adding events, adding edges, or by performing well-typed substitutions. Given a protocol Q, we say pt' refines pt, notation $pt' \subseteq pt$, iff $\mathit{traces}(Q, pt') \subseteq \mathit{traces}(Q, pt)$. Given the definition of realizable patterns, we can deduce that some patterns of a protocol can never be refined in to realizable patterns, which implies that they represent the empty trace set. The following lemmas capture typical cases in which a pattern is not contained in any trace of the protocol.

Lemma 2. *Let pt be a pattern of Q. Then we have that*

$$\big(\exists e, e', e'', t : e \to^* e' \xrightarrow{t} e'' \wedge e \neq e' \wedge t \in in(e)\big) \Rightarrow \big(traces(Q, pt) = \emptyset\big)\ .$$

Lemma 3. *Let pt be a pattern of Q, and let t be a Nonce that is sent out first at event e, i. e., according to the protocol specification, there is no earlier event than e in the same role that sends t as a subterm. Then,*

$$\big(\exists e' : e' \to^* e \wedge e' \neq e \wedge t \in in(e') \cup out(e')\big) \Rightarrow \big(traces(Q, pt) = \emptyset\big)\ .$$

4. VERIFICATION ALGORITHM

We first give an overview of the functionality of the algorithm before giving a more detailed description.

4.1 Algorithm overview

The basic idea of the algorithm is to take a pattern representing a set of traces, such as all traces violating secrecy or all traces that include an execution of a specific role. Then, this pattern is refined into a finite (possibly empty) set of realizable patterns, that represent the same set of traces as the original pattern. For properties such as checking for secrecy of a term, where the pattern can directly represent all attacks, a non-empty set of realizable patterns implies that the property is violated.

The algorithm Refine has signature

$$Protocol \times Pattern \times \mathbb{N} \to \mathcal{P}(Pattern) \cup \{used_m\}$$

and has the following functionality. Given a protocol Q, a pattern pt of Q, and an integer m, the algorithm returns a set S, i. e.,

$$\textsc{Refine}(Q, pt, m) = S,$$

such that the following three equations hold. First, all patterns in S are realizable.

$$\forall pt' \in S \cap Pattern : \text{realizable}(pt') \qquad (1)$$

Second, in case the flag $used_m$ is not present, the set of realizable patterns represents the same set of traces as the input pattern.

$$used_m \notin S \Rightarrow traces(Q, pt) = \bigcup_{pt' \in S} traces(Q, pt') \qquad (2)$$

Third, if $used_m$ is present and the terms in the events of Q contain only basic variables, any trace of the original pattern is either captured by the result, or it exceeds the parameter.

$$used_m \in S \Rightarrow \forall tr \in traces(Q, pt) : \big(tr \in \bigcup_{pt' \in S \setminus \{used_m\}} traces(Q, pt') \vee threadCount(tr) > m\big) \qquad (3)$$

The parameter m effectively limits the maximum size of the realizable patterns (in terms of the number of events), but nevertheless allows for unbounded verification, because for the vast majority of protocol patterns all possible traces are captured by a small set of realizable patterns. Furthermore, unlike e. g. CPSA or ProVerif, the algorithm always provides useful information about the absence of "small" attacks, corresponding to the verification result provided by bounded model checking tools such as OFMC [4]. Hence, in the case that $used_m$ is present, the algorithm also provides useful results, based on equation (3).

The function *threadCount* is not the only possible pruning function that can be used. It can be replaced by any function f from $Pattern \to \mathbb{N}$ that is monotonically increasing over pattern refinement by event extension as used in the algorithm, i. e. we must have that $f((E \cup F, \to)) > f((E, \to))$ each time a non-empty set of events F is added in the algorithm. For example, $f((E, \leq))$ can also be defined as the number of agent events in E, corresponding to trace length in other formalisms.

4.2 Pattern refinement

For the algorithm, we try to refine patterns into realizable patterns. If a pattern $pt = (E, \to)$ is not realizable, and $traces(Q, pt) \neq \emptyset$, there exists an event whose *in* requirements are not satisfied. Using a heuristic *selectOpen*, we pick one of these, i. e. we select a tuple (ge, gt) such that $ge \in E$, $gt \in in(ge)$ and there is no e such that $e \xrightarrow{gt} ge$. We call such a tuple an *open goal*. The heuristic influences the efficiency of the algorithm, and is described in Section 4.4.

If $traces(Q, pt)$ is not empty, then there exist patterns pt' that refine pt by some order $\to_{pt'}$ and some σ, and contain an event e such that $e \xrightarrow{\sigma gt}_{pt'} \sigma ge$, and hence $\sigma gt \in out(e)$. In case that e is a decryption event, there must exist a term $\{\!|\, t1 \,|\!\}_{t2}$ such that $\sigma gt \in unpair(t1)$. This is an essential observation for the algorithm. We can repeatedly apply case distinction until we end up at the first non-decrypt events from which gt can be inferred by using combinations of unpairing and decryption. To capture 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), we generalize the notion of unification to so-called decryption unification. We write $[\,]$ to denote the empty list.

Definition 1 (Most General Decryption Unifier). *Let σ be a well-typed substitution. We call (σ, L) a decryption unifier of a term $t1$ and a term $t2$, notation $(\sigma, L) \in DU(t1, t2)$, if either*

1. $L = [\,] \ \wedge\ \sigma(t1) \in unpair(\sigma(t2))$, *or*
2. $L = L' \cdot [\{\!|\, t \,|\!\}_k]$, $\{\!|\, t \,|\!\}_k \in unpair(\sigma(t2)) \ \wedge\ (\sigma, L') \in DU(t1, t)$.

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

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

We use the function *chain* to compute the set of most general decryption unifiers.

$$chain(t1, t2) = \big\{(\sigma, [\,]) \mid t' \in unpair(t2) \wedge \sigma = MGU(t1, t')\big\} \cup \big\{(\sigma, L \cdot [\{\!|\, t \,|\!\}_k]) \mid \{\!|\, t \,|\!\}_k \in unpair(t2) \wedge (\sigma, L) \in chain(t1, t))\big\}$$

The procedure *chain* terminates. If all variables occurring in $t2$ are basic variables, then we have that $chain(t1, t2) =$

$MGDU(t1,t2)$. If $t2$ contains non-basic variables, *chain* might not cover all options. We mitigate this problem in the majority of cases by application of Lemma 2, and the choice of heuristics in the algorithm. In cases where these both fail for a given protocol, the results are marked incomplete, as in the bounded verification case, and the remaining finite possibilities are unfolded similar to [5]. Unlike [5], here we only need to consider the finite case for unfolding of encryptions.

If we add protocol events to the pattern, we must ensure the pattern meets the protocol consistency requirements. For this purpose we define an auxiliary function *prefixClose*, which ensures for all events that the preceding events from that same role also occur in the same thread.

Algorithm 1 Refine(Q,pt,m)

```
Require: $Q$ is a protocol, $pt$ is a pattern, and $m$ is an integer.
Ensure: Returns a set of realizable patterns. The set may include the special symbol '$used_m$' when the complete set of realizable patterns can not be determined.
  if $threadCount(pt) > m$ then
    return $\{used_m\}$
  else
    if $prune(Q,pt)$ then // Prune checks for the preconditions of Lemma 2 and 3.
      return $\emptyset$
    else
      if $realizable(Q,pt)$ then
        return $\{pt\}$
      else // $pt$ is not realizable.
        $(ge,gt) \Leftarrow selectOpen(pt)$ // Apply heuristic to select an unbound term, and apply case distinction on possible earliest bindings of $gt$.
        $result1, result2, result3 \Leftarrow \emptyset, \emptyset, \emptyset$
        for all $(e,C) \in \{(e, chain(t,gt)) \mid e \in E_{pt} \wedge t \in out(e)\}$ do
          $pt' \Leftarrow \sigma_C(pt \cup \{e \xrightarrow{C_n} \cdots \mathsf{decr}(C_i) \cdots \xrightarrow{gt} ge\})$
          $result1 \Leftarrow result1 \cup \textsc{Refine}(Q,pt',m)$
        end for
        if $gt = \{\!| t1 |\!\}_{t2}$ then
          $pt' \Leftarrow pt \cup \{\mathsf{encr}(gt) \xrightarrow{gt} ge\}$
          $result2 \Leftarrow \textsc{Refine}(Q,pt',m)$
        end if
        for all $(pt',e,C) \in \{(pt',e,chain(t,gt)) \mid t \in out(e) \wedge e \in ev(Q) \wedge pt' \in prefixClose(Q,e)\}$ do
          if $e$ is not an init event then
            $newid \Leftarrow$ a thread identifier that does not occur in $pt$.
            for all $x \in threadIDs(pt) \cup \{newid\}$ do
              $\sigma' \Leftarrow \sigma_C \cup [x/\tau]$
              $pt'' \Leftarrow \sigma'(pt \cup pt' \cup \{e \xrightarrow{C_n} \cdots \mathsf{decr}(C_i) \cdots \xrightarrow{gt} ge\})$
              $result3 \Leftarrow result3 \cup \textsc{Refine}(Q,pt'',m)$
            end for
          end if
        end for
        return $result1 \cup result2 \cup result3$
      end if
    end if
  end if
```

The resulting procedure Refine, shown as Algorithm 1, is guaranteed to terminate. Each iteration either decreases the number of terms in the $in(e)$ set of an event e which have no corresponding incoming edges, or increases the number of honest agent events. From these two elements we can construct a non-increasing invariant (measure) that ensures termination.

The correctness of unbounded verification, corresponding to Formula (2), depends on the algorithm exploring all possibilities for enabling the receive events.

Theorem 1. *Let pt be a pattern of a security protocol Q, and let m be an integer. Let $S = \textsc{Refine}(Q,pt,m)$ such that $used_m \notin S$. Then*

$$traces(Q,pt) = \bigcup_{pt' \in S} traces(Q,pt').$$

Proof sketch: If the refinement algorithm returns a non-empty set S of realizable patterns, it is straightforward to see that any trace of one of these realizable patterns also exhibits the original pattern pt, based on the notion of pattern refinement. The converse, that all traces of pt are captured in S, depends on the observation that any trace that exhibits pt must be realizable. Let t be a trace of $traces(Q,pt)$. The algorithm refines the pattern pt for all possible traces of Q, such that one of the refinements must contain t. In particular, all receive events in t must be enabled by a (set of) preceding events. These preceding events must have non-empty *out* sets, and hence must be either send, decr, encr, or init events. Branching is done on the type of event, and for all possible thread identifiers. These include the thread identifiers occurring in the pattern as well as a fresh (i. e. not occurring in the pattern yet) thread identifier. In the case that a decr event is assumed to be the enabling event, there must exist a finite chain of decrypt events, preceded by a non-decrypt event.

4.3 Verifying security properties using pattern refinement

Next we show how the algorithm is used for verification, falsification and characterization.

Verification and falsification of confidentiality properties.

Confidentiality properties are commonly of the form “for all traces of the protocol, if an honest agent executes a role R of a protocol Q, and the intended communication partners are honest, then the intruder should never learn a particular term”. We construct a pattern pt that captures exactly all traces that violate this confidentiality property. The pattern represents all traces in which the property is false, and therefore it consists of the following:

1. the events of the role R, along with the order induced by the role specification, and including only honest agent names (captured by the typing system, by substituting the agent variables by variables that may be substituted only by honest agent names),
2. the initial intruder knowledge, including the private keys of compromised agents, represented by init events,
3. the event $\mathsf{know}(t)$ that expresses the intruder knows t at some point.

In a pre-processing step of the algorithm, security properties are automatically turned into a pattern that represents all traces that violate the property.

Next, the refinement algorithm is applied to Q, pt, m for some m and returns the result set S. We distinguish three cases.

1. $S \cap Pattern \neq \emptyset$: Following Lemma 1 we can directly construct traces of Q that exhibit pt from each element of $S \cap Pattern$, which represent attacks. Hence secrecy is violated, and the property is falsified.

2. $S = \emptyset$: From equation (2) we have that there exist no traces exhibiting the pattern, hence no attacks exists. This constitutes unbounded verification.

3. $S = \{used_m\}$: Based on equation (3) we conclude that there are no attacks with m or less threads, constituting bounded verification. Increasing m may yield unbounded verification or falsification.

Characterization of security protocols.

Characterization, as described in [16], provides a concise finite representation of all possible protocol behaviours. The approach essentially consists of giving for all roles a finite representation of all traces that include an instance of that role. This corresponds directly to the functionality of the algorithm presented here, and can be performed by applying the algorithm to a pattern that consists of (1) the events of the role R, with the order induced by the role specification, and including only honest agent names, (2) initial intruder knowledge events init. The result of the algorithm, if it does not contain $used_m$, provides a complete characterization. If the result contains $used_m$, it only characterizes all traces which can be represented by realizable patterns that do not have more than m threads. An example of a complete characterization in our algorithm is provided in Appendix A.

Note that the characterizations generated by our refinement algorithm differ slightly from those in [16]. We elaborate on the differences in Section 5.1.

Verification and falsification of authentication properties.

We first apply the characterization process and then check whether the authentication property holds for each realizable pattern. This allows for verification of e. g. aliveness, non-injective agreement. It also allows us to efficiently establish ordering related properties such as non-injective synchronisation.

4.4 Heuristics

The heuristic *selectOpen* used in the algorithm influences both the effectiveness and efficiency of the algorithm. Recall that an open goal is a tuple (ge, gt), where $gt \in in(ge)$, that needs to be connected by an incoming edge labeled with gt in order to arrive at a realizable pattern. The heuristic selects one of possibly many such open goals, which is used for case distinction and pattern refinement. Although the algorithm will try to bind any other open goals in further iterations, any substitutions made by the case distinctions and refinement steps influence the branching factors further on. Furthermore, for some heuristics, contradictory states (corresponding to patterns with empty trace sets) may occur earlier in the iteration process. 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. Note that a similar heuristic must exist in the Athena algorithm [28], but has not been elaborated by the authors.

As our main goal is to establish falsification or unbounded verification, we choose a heuristic that is optimal for the *effectiveness* of the algorithm: to achieve, even with a low choice for the parameter m, unbounded verification for as many protocols as possible. Note that this may be less efficient for particular protocols where unbounded verification is not achieved (and hence cause the algorithm to be slower).

Observe that the choice of heuristic influences efficiency and effectiveness of the algorithm, but it does not influence the correctness of the algorithm.

We devised over 20 candidate heuristics and investigated their effectiveness. Here we report our main findings and illustrate them by means of a few selected heuristics, ordered according to their effectiveness.

- *Heuristic 1: Random.* An open goal is selected randomly for case splitting.
- *Heuristic 2: Constants.* For each open goal term t, the number of local constants that are a subterm of t, is divided by the number of basic terms that are a subterm of t. The goal with the highest ratio is selected.
- *Heuristic 3:* Open goals that correspond to the keys needed for decrypt events are given higher priority, unless these keys are in the initial intruder knowledge.
- *Heuristic 4:* Give priority to goals that contain a private key as a subterm; next, give priority to goals that contain a public key ; all other terms have lower priority.
- *Heuristic 5:* A combination of heuristics 2, 3 and 4, where first heuristic 4 is applied. If this yields equal priorities for a goal, heuristics 2 and 3 are applied.

Regarding heuristic 4, we observe that not all security protocol semantics explicitly mention such concepts as 'private' or 'public' key (there might be no such terms, or multiple key infrastructures). We derive these from the initial intruder knowledge and role descriptions by identifying function names which are never sent as a subterm, but only as keys. This will typically include the asymmetric key functions sk and pk. Second, we observe that for some functions, all applications are in the initial intruder knowledge, usually the public keys such as pk, whereas for others only a strict subset of the domain is part of the initial intruder knowledge, usually the private keys sk.

For all heuristics, we have that if two open goals are assigned the same priority value, the open goal that was added first is selected. Tests have shown this to be slightly more effective for all heuristics involved.

The first heuristic acts as a reference point for establishing relative effectiveness of each heuristic. The second heuristic corresponds to the intuition that terms which contain more local constants of particular threads, can only be bound to very particular send events (as opposed to terms with many globals or variables), resulting in less case distinctions. We believe a similar heuristic was used in a version of the Athena tool. The third heuristic captures the intuition that there should be few ways in which the intruder can gain access to a decryption key, as in general keys should not be known to the intruder. (Unless it concerns signatures, in which case the decryption key is the public key, which is part of the initial intruder knowledge.) For the fourth heuristic, a strict priority is given to cases where e. g. the intruder decrypts something with a key that is never sent by the regular agents, usually corresponding to long-term keys, as these branches often lead to contradictory states. Finally, the fifth heuristic is a combination of the previous three heuristics, using a lexicographic order. For the fifth heuristic various weighting functions were also considered, of which the lexicographical order performed best in general.

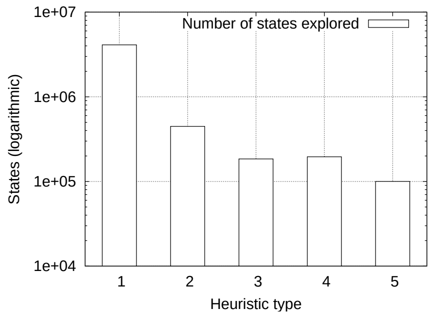

Figure 1: The impact of the heuristics on the efficiency (number of states traversed, for 518 claims)

Given a fairly low setting of the parameter, in particular we set $m = 4$, we have investigated how each heuristic performed, when applied to a test set of 128 protocol descriptions, with 518 security claims. The test set includes the vast majority of the protocols in the SPORE library [29], various protocols from scientific papers, some variations on existing protocols, and new protocols, as modeled by users of the Scyther tool. A time limit was set for the iteration procedure, which was only used to abort tests for the first two heuristics. In Figure 1 we show the impact of the heuristics on the number of states explored. From the graph it is clear that heuristic 5 explores almost 40 times less states than the random heuristic 1. Intuitively, this corresponds to avoiding unnecessary branching, and a tendency to arrive at contradictory trace patterns in less iterations.

Because the effectiveness of the heuristics depends to a large degree on the particular protocol under investigation, it is difficult to give an analytical explanation of the results for the complete test set. However, it seems that the heuristics 2, 3 and 4 can be used to support each other, as is shown by the performance of heuristic 5.

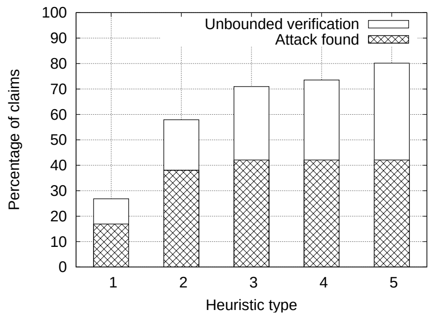

Figure 2: The impact of the heuristics on the effectiveness (for 518 claims, $m = 4$)

The heuristic also has a direct result on the completeness of the results, which is depicted in Figure 2. For heuristic 1, we get a complete result (based on complete characterization) for less than 30 percent of the claims. This improves for each heuristic, leading to an 82 percent rating for heuristic 5. In other words, if we use heuristic 5, we have that for 82 percent of the claims in the test set, the algorithm is able to either find an attack, or verify correctness for an unbounded number of threads. In the remaining 18 percent of the cases the algorithm determines that there are no attacks involving four threads or less, but it might be possible that there are attacks involving five or more threads. We conclude that heuristic 5 is to be preferred from the set of investigated heuristics.

5. RELATED WORK AND PERFORMANCE

5.1 Related algorithms and tools

Although there are a large number of algorithms and tools for the bounded verification of protocols, e. g. [1, 4, 11, 22], there are only a few methods suitable for unbounded verification. The techniques used for unbounded verification are usually based on overapproximations of the protocol execution model. This includes using methods based on static typing, e. g. [9,19], overapproximation of nonce generation [6,8] , or more recently by causality-based abstraction of multiplicity [2,3]. The methods based on overapproximation are often efficient for protocols in which each protocol message can be uniquely identified, e. g. in the case for protocols that are sufficiently tagged for methods based on typing information. On the other hand, these methods commonly suffer from two problems. First, except for ProVerif, these methods in general do not allow for the generation of counterexamples (attacks), and their scope is restricted, in the sense that the techniques often fail for common protocols that are not sufficiently tagged. The current state-of-the-art in unbounded verification based on overapproximation is ProVerif, which can generate valid counterexamples in some cases when the unbounded verification fails. However, all these methods have cases in which the algorithms fails to produce any results, i. e. in which the tool can make no statement about the property, or does not terminate.

The algorithm presented here is most closely related to the Athena algorithm as described in e. g. [28], but the associated tool has never been publicly released. The Athena algorithm has been reported to be efficient. However, the algorithm as described in [28] suffers from the requirement of untyped variables (which is needed to model the intruder roles in the strand spaces). As a result, the set of "interm" unifiers is by definition infinite for the intruder roles, which causes infinite branching at each "split" step of the Athena algorithm. This problem has been acknowledged by Berezin in [5]. Furthermore, the strand space framework underlying the Athena tool does not allow for composed keys, and the algorithm does not allow for verification of ordering related security properties such as synchronisation.

In [16] a method is developed to compute the *complete characterization* of a protocol role, using an extended version of the strand spaces model. In this work, characterizations correspond to ordered sets events of honest agents only, with information-preserving homomorphisms and confidential keys. The homomorphisms are an alternative way of dealing with both equivalence up to renaming and variable instantiation. The confidential keys are an alternative way of expressing honesty assumptions on agents. The characterizations of CPSA have three main drawbacks. First, not all homomorphisms (or variable instantiations) may lead to executable traces, and they therefore correspond to an over-approximation of realizable patterns. Second, the strong connection to authentication tests means that protocols that can not be analysed using authentication tests, cannot be characterized, even when they may have meaningful characterizations. (As a trivial example, consider a possibly flawed protocol that does not use encryption at all.) Third, the CPSA algorithm is not guaranteed to terminate. Furthermore, to the best of our knowledge CPSA cannot provide a complete characterization in cases where Scyther can provide it, e. g. for the Otway-Rees protocol, even though this protocol should fall into the domain of protocols that CPSA can characterize.

5.2 Performance

The refinement algorithm implemented in Scyther provides state-of-the-art performance in terms of efficiency. In Table 2 we provide an impression of the performance of the Scyther tool. In the table, the NSPK, NSPK-FIX and Otway-Rees protocols has been modeled according to SPORE [29]. The two versions of the TLS protocol are taken from a paper [27] and the AVISPA library of protocols [20]. The $f^n g^n$ protocols are instances of a family of protocols described in [25]. The protocols from this family are not intended for practical usage, but are explicitly designed to exhibit an attack involving $n + 1$ threads, but no attack with fewer threads.

Table 2: Verification times (1.66 GHz Intel Centrino processor, 1GB ram, Linux)

Protocol	Time	Details
NSPK	0.1s	attack
NSPK-FIX	0.1s	verified
Otway-Rees	0.1s	verified (typed variables)
Otway-Rees	0.0s	attack (type flaw)
TLS (Paulson)	0.2s	verified
TLS (Avispa)	0.2s	attack (“Alice talks to Alice”)
NSPK-FIX || NSPK-alt	0.3s	attack (multi-protocol attack)
$f^{10}g^{10}$	0.2s	attack using 11 threads
$f^{30}g^{30}$	10.1s	attack using 31 threads
$f^{50}g^{50}$	110.1s	attack using 51 threads

The efficiency of the algorithm has made new types of protocol analysis possible, including multi-protocol analysis as performed in [12]. In such an analysis, the assumption that a protocol is the only protocol running in the environment is dropped. Because security properties do not compose in general, it can be the case that a security protocol that is correct in isolation, becomes incorrect when composed (e. g. sequentially, or in parallel) with another security protocol. However, the complexity of such an analysis, when compared to traditional (single) protocol analysis, is exponential in the number of composed protocols. Using a version of the refinement algorithm presented here, the author of [12] was able to efficiently perform large scale multi-protocol analysis tests involving up to three composed protocols, which was not feasible with any other tool.

The performance of the Scyther tool has also been compared to a number of other state-of-the-art protocol verification tools, including ProVerif, and the four AVISPA tools: TA4SP [8], OFMC [4], Cl-AtSe [31], and SatMC [1]. These tests are described in [14]. In the tests, Scyther outperformed the bounded tools, OFMC, Cl-AtSe, and SatMC. As a result, to the best of our knowledge, Scyther is the fastest available tool that does not use approximation techniques.

Two of the compared tools in [14] can also perform unbounded verification. These are ProVerif and TA4SP. In our tests, ProVerif and Scyther often perform similarly. In the single case where Scyther cannot perform unbounded verification, its execution time is exponential with respect to the parameter m. Both Scyther and ProVerif outperform TA4SP both in speed as well as in scope.

Two other related tools, Athena [28] and CPSA [16], have not been publicly released. We are therefore not able to provide any performance comparisons. The minimal sets of reported execution times in papers suggest that Athena has similar execution times to Scyther, whereas CPSA seems significantly slower.

6. FUTURE WORK

As future work, it will be of interest to investigate how algebraic properties can be efficiently integrated in the algorithm, allowing for handling of e. g. exclusive-or or modular exponentiation. If we restrict the algebraic properties such that the set of unifiers of two terms is finite, integration is straightforward. However, if the set of unifiers is infinite, additional techniques are required to extend our algorithm to include these algebraic properties.

As already suggested in [28], the exploration of the search space by backwards search may serve as a useful basis for applying automatic theorem proving. This would turn the unbounded verification result of the tool in to a machine-checked proof of correctness. (Note that this would only be a proof of the security property of the protocol model in the abstract execution model, not an “absolute” proof of security.) However, for excluding bugs in the verification tool, as well as certification purposes, this would be a valuable result. Work in this direction has already started [24].

Other future directions include strengthening the Dolev-Yao intruder model underlying the verification mechanism along the lines of e. g. guessing attacks.

7. CONCLUSIONS

We have presented an algorithm for the unbounded verification, falsification, and characterization of security protocols. The algorithm provides a number of novel features as well as state-of-the art performance.

The algorithm builds on ideas from [28]. Our contributions include a generalized algorithm that results in a wider scope of protocol models that can be handled. By improving pruning rules and a different way to deal with the intruder events, we are able to deal with variables that can contain tuples and encryptions. We provide a means to ensure termination whilst still proving a meaningful verification result (i. e. bounded verification), and we allow for verification of a larger class of security properties.

Our contributions also include an alternative way to generate complete characterizations as in [16], not only providing a very efficient way to generate characterizations, but also for a larger class of protocols than handled by CPSA. Our reformulation of characterization in terms of realizable patterns is independent of concepts such as authentication tests.

The implementation of the algorithm in the Scyther tool provides state-of-the-art performance, shown in [14]. In combination with the semantics from [15], it has made novel types of analysis feasible, such as multi-protocol analysis [12].

8. REFERENCES

[1] A. Armando and L. Compagna. Sat-based model checking for security protocols analysis. *International Journal of Information Security*, 7(1):3–32, 2008.

[2] M. Backes, S. Lorenz, M. Maffei, and K. Pecina. The CASPA tool: Causality-based abstraction for security protocol analysis. In A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 419–422. Springer, 2008.

[3] M. Backes, M. Maffei, and A. Cortesi. Causality-based abstraction of multiplicity in security protocols. In *Proc. 20th IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, June 2007.

[4] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[5] 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` (unpublished manuscript), 2001.

[6] 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.

[7] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[8] Y. Boichut, P.-C. Héam, O. Kouchnarenko, and F. Oehl. Improvements on the Genet and Klay technique to automatically verify security protocols. In *Proc. International Workshop on Automated Verification of Infinite-State Systems (AVIS'2004)*.

[9] M. Bugliesi, R. Focardi, and M. Maffei. Authenticity by tagging and typing. In *Proc. 2nd ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 1–12. ACM Press, 2004.

[10] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

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

[12] C. 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.

[13] C. Cremers. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, Proc.*, volume 5123/2008 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.

[14] C. Cremers and P. Lafourcade. Comparing state spaces in automatic protocol verification. In *Proc. of the 7th Int. Workshop on Automated Verification of Critical Systems (AVoCS'07)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Direct, September 2007.

[15] C. Cremers and S. Mauw. Operational semantics of security protocols. In *Scenarios: Models, Transformations and Tools, International Workshop, 2003, Revised Selected Papers*, volume 3466 of *Lecture Notes in Computer Science*. Springer, 2005.

[16] S. Doghmi, J. D. Guttman, and F. Thayer. Skeletons, homomorphisms, and shapes: Characterizing protocol executions. In *Proc. of the 23rd Conf. on the Mathematical Foundations of Programming Semantics (MFPS XXIII)*, volume 173 of *Electronic Notes in Theoretical Computer Science*, pages 85–102. Elsevier ScienceDirect, April 2007.

[17] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(12):198–208, Mar. 1983.

[18] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Proc. of the FLOC'99 Workshop on Formal Methods and Security Protocols (FMSP'99)*, 1999.

[19] C. Haack and A. Jeffrey. Pattern-matching spi-calculus. *Inf. Comput.*, 204(8):1195–1263, 2006.

[20] F. O. P. IST-2001-39252. AVISPA: Automated validation of internet security protocols and applications, 2003.

[21] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[22] G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW)*, pages 18–30. IEEE Computer Society, 1997.

[23] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[24] S. Meier. A formalization of an operational semantics of security protocols. Diploma thesis, ETH Zurich, August 2007. `http://people.inf.ethz.ch/meiersi/fossp/index.html`.

[25] J. Millen. A necessarily parallel attack. In *Workshop on Formal Methods and Security Protocols*, Trento, Italy, 1999.

[26] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[27] L. Paulson. Inductive analysis of the Internet protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, Aug. 1999.

[28] 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.

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

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

[31] M. Turuani. The CL-Atse protocol analyser. In *Proc. RTA'06*, volume 4098 of *Lecture Notes in Computer Science*, pages 227–286. Springer, Aug. 2006.

APPENDIX

A. THE SCYTHER TOOL

An instance of the algorithm presented in this paper was implemented in the Scyther tool [13]. The tool uses a security protocol model with operational semantics from [15], and can verify security properties such as secrecy of terms and various authentication properties, including synchronisation [15]. Additionally, the tool can establish complete characterization [16] of protocol roles. The tool is freely available for Windows, Linux, and Mac OS X, and can be downloaded from `http://people.inf.ethz.ch/cremersc/scyther/`.

The input in the left of Figure 3 shows the description of the Needham-Schroeder protocol, including its security properties, which can be loaded by the Scyther tool. The input format is essentially a role-based description of a protocol, and is based on the operational semantics for security protocols as defined in [15]. In this formalism, security properties are denoted by so-called *claim events*. These are considered to be part of the protocol description. The Scyther tool can establish whether these claims are satisfied (verification) or not (falsification).

```
// 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);
```

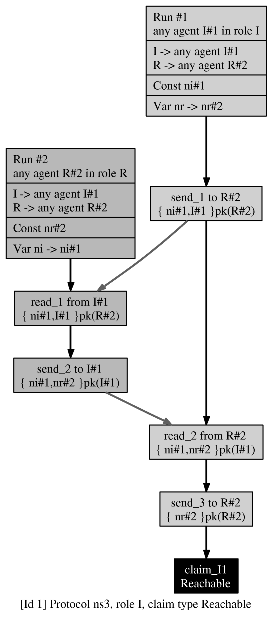

Figure 3: Left: Input file for the Scyther tool, describing the Needham-Schroeder protocol. Right: Role I complete characterization: a single pattern.

In order to give an impression of what a complete characterization looks like, we show a complete characterization of the two roles of the Needham-Schroeder protocol.

The complete characterizations are produced automatically by the Scyther tool. In particular, using the graphical user interface of Scyther, the characterization graphs can be reproduced by simply loading the file `ns3.spdl` (shown in the left of Figure 3) and pressing the "Characterize roles" button found in the "Verify" menu. This produces the right of Figure 3 and Figure 4.

In the graphs, the boxes and ellipses represent events. Vertical downwards arrows, which are unlabeled, represent the ordering of events within a thread (or "run" in these graphs). Thread identifiers have a '#' prefix. Boxes represent events executed by honest agents, or elucidate the details of a thread, if they are the first box on a thread. Ellipses represent intruder events, or are used to elucidate implicit intruder behaviour (such as "fake sender").

For the initiator role of the Needham-Schroeder protocol, there is only one trace pattern, shown in the right of Figure 3. Thus, all traces that include the initiator role, must also include the structure in the graph, which exactly corresponds to a valid protocol execution. As a result, any authentication claim (including synchronisation) at the end of the initiator role is correct.

For the responder role, there are exactly two explicit trace patterns, shown in Figure 4. 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.

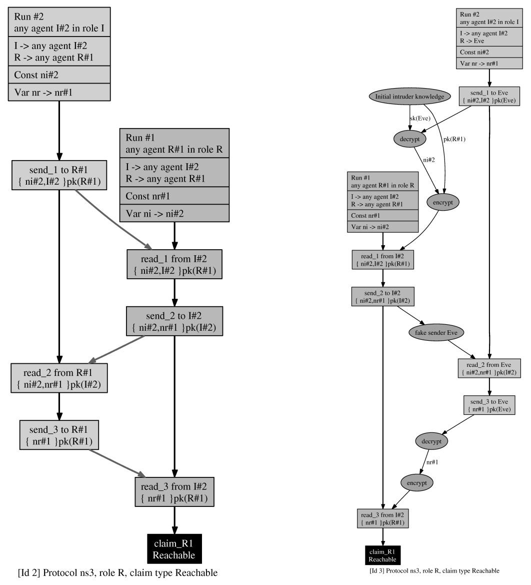

Figure 4: Needham-Schroeder, role R complete characterization: two patterns.

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.