Customizing Protocol Specifications for Detecting Resource Exhaustion and Guessing Attacks*

Bogdan Groza and Marius Minea

Politehnica University of Timişoara and Institute e-Austria Timişoara
bogdan.groza@aut.upt.ro, marius@cs.upt.ro

Abstract. Model checkers for security protocols often focus on basic properties, such as confidentiality or authentication, using a standard model of the Dolev-Yao intruder. In this paper, we explore how to model other attacks, notably guessing of secrets and denial of service by resource exhaustion, using the AVANTSSAR platform with its modelling language ASLan. We do this by adding custom intruder deduction rules and augmenting protocol transitions with constructs that keep track of these attacks. We compare several modelling variants and find that writing deductions in ASLan as Horn clauses rather than transitions using rewriting rules is crucial for verification performance. Providing automated tool support for these attacks is important since they are often neglected by protocol designers and open up other attack possibilities.

1 Introduction and Motivation

Formal verification tools provide an efficient means for automatic verification of security protocols, once models of these have been written, e.g., some variant of symbolic transition systems. Usually, the focus is on verification of standard security goals, such as authenticity and confidentiality. However, in many cases, satisfying these goals is not sufficient to consider a protocol safe and a more in-depth analysis to rule out other kinds of attacks is necessary.

This paper focuses on two such attacks which are not handled routinely by many protocol verifiers, namely guessing attacks and denial of service (DoS). Both of these attacks are a main concern in protocol design. Guessing attacks are relevant because users tend to choose weak passwords, and some values such as PIN codes have intrinsically low entropy. They can become the weakest link in more complex protocols, leading to other attacks as well. Resource exhaustion is relevant as a common source of DoS as well as from an economic point of view if one considers ruling out protocol designs that can be exploited to make honest participants spend unreasonable amounts of resources, time or memory.

* This work is supported in part by FP7-ICT-2007-1 project 216471, AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures and by strategic grant POSDRU/21/1.5/G/13798 of the Human Resources Development Programme 2007-2013, co-financed by the European Social Fund – Invest in People.

B.K. Aichernig, F.S. de Boer, and M.M. Bonsange (Eds.): FMCO 2010, LNCS 6957, pp. 45–60, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Our research is performed in the framework of the AVANTSSAR project, where security protocols and services are specified as transition systems in the ASLan specification language, based on set rewriting. ASLan models can be analyzed by three different model checkers as back-ends: CL-Atse [19], OFMC [4] and SATMC [2]. None of them handles DoS or guessing attacks by default.

We present several ways to model custom deduction rules in ASLan. The purpose is to devise a way to augment protocol models so they can be analyzed for guessing and DoS attacks without changing the model checkers.

To introduce customized attack rules at the ASLan level there are two main options: adding new intruder transitions and/or adding Horn clauses. The first approach results in mixing protocol steps with the new customized transitions, which can significantly increase state space explosion and verification time. Using Horn clauses is more efficient but faces several issues. both due to the ASLan design (Horn clauses cannot add to intruder knowledge, which would be needed in guessing) and because the model checkers differ in their level of support for Horn clauses and in their search strategies. Consequently, to express customized attack rules, we need to write additional transition rules as well as Horn clauses.

Section 2 presents the ASLan specification language focusing on the main features relevant for our modelling. In Section 3, we briefly present the principles behind our analysis of DoS attacks and then details of their modelling with customized transitions. Section 4 describes the modelling of guessing attacks, with several versions using customized transitions and, for efficiency, Horn clauses for intruder deductions. We conclude in Section 5 with a discussion of the results.

2 The ASLan Specification Language

The AVANTSSAR specification language ASLan is an expressive language for specifying security protocols and services as well as their policies, based on set rewriting. In the following, we give a simplified account of the language, focusing on the features which are most relevant for our customized modelling of attack rules. A full description of the language is given in [3].

ASLan models are transition systems in which each state is modeled by a set of ground facts. Predefined types include `message` and its subtypes (`agent`, `private_key`, `public_key`, `symmetric_key`, `text`). The user can declare additional type symbols, functions and facts (predicates) with their type signatures.

For example, consider the MS-CHAP protocol, a known target for guessing attacks. Figure 1 presents the description in Alice-and-Bob notation, together with an ASLan transition rule for role A, who on receiving nonce N_B in step 2 responds with N_A and a hash computation in step 3.

Here, `state_A` is a fact that tracks the state of principal A, including an instance identifier `ID`, a step counter that changes from 1 to 2, and other known values, including the identity of `B`, the shared key, the hash function, and the two nonces (which become known as a result of the step). Communication is modeled using the fact `iknows` (on the left-hand side for receive, and on the right-hand side for send), since anything transmitted becomes part of the intruder knowledge. Conjunction of facts is represented by a dot; `apply` represents function

application and `pair` message concatenation. The `exists` keyword specifies the creation of a fresh value as part of the transition.

1. $A \rightarrow B : A$
2. $B \rightarrow A : N_B$
3. $A \rightarrow B : N_A, H(k_{AB}, N_A, N_B, A)$
4. $B \rightarrow A : H(k_{AB}, N_A)$

```
step step_1(A,B,H,ID,Kab,Na,Nb,Na0,Nb0) :=
  state_A(A,ID,1,B,Kab,H,Na0,Nb0)
  .iknows(Nb)
  =[exists Na]=>
  state_A(A,ID,2,B,Kab,H,Na,Nb)
  .iknows(pair(Na,apply(H,
          pair(Kab,pair(Na,pair(Nb,A))))))
```

Fig. 1. MS-CHAP v2 protocol and ASLan transition rule

Let $\mathcal{F}$ be the set of ground facts; the set of all possible states is then $\mathcal{S} = 2^{\mathcal{F}}$. An ASLan model defines a transition system $M = \langle \mathcal{S}, \mathcal{I}, \rightarrow \rangle$, where $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states and $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation.

In an ASLan model, the set of initial states is a conjunction of facts. Transitions are rewrite rules where both sides are conjunctions of facts. A transition can be taken from any state that contains the facts on the left-hand side; these are removed from the state and replaced by the facts on the right-hand side. As an exception, `iknows` (intruder knowledge) is a persistent fact and does not disappear, even if written on the left-hand side and being omitted on the right.

Formally, we first define the closure $\lceil S \rceil^H$ of a state S with respect to the set H of Horn clauses in the model as the set of all ground facts that can be derived from S using H. More precisely, $\lceil S \rceil^H$ is the smallest set containing S such that

$$\forall F \leftarrow F_1, \cdots, F_n \in H, \forall \sigma. \bigcup_{1 \leq i \leq n} F_i\sigma \subseteq \lceil S \rceil^H \Rightarrow F\sigma \in \lceil S \rceil^H$$

where σ is any substitution function that maps the variables of the Horn clause $F \leftarrow F_1, \cdots, F_n$ to ground terms.

A transition rule in ASLan has the form $PF.\ NF\ =[V]=>\ R$, where PF is a set of positive facts, NF is a set of negative (negated) facts, V is a set of fresh introduced variables, and the right-hand side R is a conjunction of facts.

We can now define the transition relation $\rightarrow$ as follows: there is a transition $S \rightarrow S'$ iff there exists a transition rule $PF.\ NF\ =[V]=>\ R$ and a substitution σ from the variables of PF to ground terms such that following conditions hold:

- $PF\sigma \subseteq \lceil S \rceil^H$, i.e., the positive facts on the left-hand side hold in $\lceil S \rceil^H$
- $NF\sigma\sigma' \cap \lceil S \rceil^H = \emptyset$ for all substitutions σ' such that $NF\sigma\sigma'$ is ground, i.e., the negative facts cannot hold in $\lceil S \rceil^H$
- $S' = (S \setminus PF\sigma) \cup R\sigma\sigma''$, where σ'' is any substitution such that for all $v \in V$, $v\sigma''$ does not occur in S (i.e., variables in V are substituted with fresh terms).

The combination of transition rules and Horn clauses in the language implies the existence of two kinds of facts. *Explicit* facts are introduced by the right-hand side of transition rules and are persistent unless removed by a later transition (if present on the left-hand side but not the right-hand side). *Implicit* facts are introduced by Horn clauses and are recomputed as part of the state closure after each transition step. To ensure a consistent semantics, explicit facts (including

the intruder knowledge `iknows`) cannot appear in the conclusion of a Horn clause. This impacts our design of guessing rules, which must add intruder knowledge.

These definitions lead to an execution model for an ASLan specification that alternates Horn clause deductions and transition steps: first, the set of facts comprising a state is augmented by the facts obtained by performing the transitive closure of the Horn clauses, and then one of the applicable transition rules is chosen and executed, after which the entire process is repeated. In particular, this makes Horn clauses suitable for modelling intruder deduction and any additional processing necessary for attack detection, as Horn clause deductions are performed after each transition step.

3 Customized Transitions for Detection of DoS Attacks by Resource Exhaustion

We formalize costs and attack conditions in order to detect DoS attacks by resource exhaustion. While we focus on computation resources due to the varying cost of cryptographic primitives, costs could be associated to memory consumption or other resources as well.

Resource exhaustion DoS attacks can be divided according to the behaviour of the adversary in two categories: one is *abusive* use of the service by clients which willingly or not deplete the server from resources, the other is *malicious* use in which adversaries manipulate protocol messages and make honest principals waste computational time without reaching protocol goals. For the first case, we consider an attack feasible if the initiator can force repeated use of the protocol, which leads to resource depletion. For the second case we consider the protocol under attack when principals reach states in which their beliefs about the protocol are wrong, e.g., messages are accepted from impersonated senders. Cutting down communication is not an issue since the intruder can do this for any protocol and protocol design cannot give countermeasures to it. In both cases, to deem a resource exhaustion attack successful we must evaluate costs for both the adversary and honest principals. An attack is flagged as successful when both the cost of the adversary is lower and one of the two situations hold: the adversary is the initiator or the principal's beliefs are wrong.

3.1 Defining Costs and Augmenting Transitions

Costs can be treated according to the framework of Meadows [16], which uses a monoid structure; this approach is also used in follow-up related work [17,18]. The cost set employed is $S = \{0, low, medium, high\}$, and the sum of two costs is simply defined as their maximum: $\forall a, b \in S, a + b = max(a, b)$. This can be easily modeled in ASLan by using a fact for summing costs, as shown in Figure 2 where cost values are of type `text` and `sum` has the signature `sum: text * text * text -> fact`. In the same manner, the comparison between cost values is defined with the fact `less`.

The existing AVANTSSAR model checkers have limited support for numeric values. Using SMT-based techniques would allow for integer costs and a better evaluation of complex attacks such as distributed DoS, where a more sensitive cost analysis must be done. For example, the initial cost of the adversary can be high, but it can be alleviated over multiple protocol sessions. Only a few manual analyses have been done with explicit numeric cost values [15]; most analyses in the literature are symbolic, using a monoid as cost structure.

Transitions can be easily augmented by costs. This has to be done for both protocol steps (as described in detail in [11]) and intruder deductions. Figure 2 shows the cost definition and an intruder deduction modeled as protocol transition that keeps track of cost. The example is a deduction in which the intruder performs a signature with key Y over term X, denoted by `costSig(X,Y)`, and incurring cost `high`, with the initial condition that he knows both X and Y.

```
sum(low, low, low).
sum(low, medium, medium).
sum(medium, low, medium).
sum(low, high, high).
sum(high, low, high).
sum(high, medium, high).
sum(medium, high, high).
sum(high, high, high).
less(low, medium).
less(medium, high).
less(low, high)
```

```
step trans_1(X, Y, Cost, NewCost, ID):=
  state_adv(i, ID, 0)
  .iknows(X).iknows(Y)
  .cost(i, Cost)
  .sum(Cost, high, NewCost)
  =>
  state_adv(i, ID, 0)
  .iknows(costSig(X,Y))
  .cost(i, NewCost)
  .sum(Cost, high, NewCost)
```

Fig. 2. Defining costs (left) and a cost-augmented transition for a signature (right)

3.2 Defining the Attack Condition

To flag an attack on principal P, a necessary condition is that the intruder cost is less than the cost incurred by P: $\texttt{cost}(\texttt{i}, \texttt{C}_\texttt{i}).\texttt{cost}(\texttt{i}, \texttt{C}_\texttt{P}).\texttt{less}(\texttt{C}_\texttt{i}, \texttt{C}_\texttt{P})$. In addition, for abusive use, we need to keep track of the protocol initiator. This can be done by augmenting the initial transition of the protocol (done by principal A) with the fact `initiate(A)` and adding `initiate(i)` in the attack condition (the attack must be repeatable by the intruder). For malicious use, we track the violation of injective agreement. This can be done by augmenting the right-hand side of each send and receive transition with the facts `send(S, R, M, L, ID)` and respectively `recv(S, R, M, L, ID)`, having as parameters the sender, recipient, content, protocol step and instance. The attack is flagged by checking satisfiability of the condition `recv(S, R, M, L, ID).not(send(S, R, M, L, ID))`, which means that a message receive does not have a matching send.

These attack conditions can be further refined along other criteria, such as determining whether the attack is detectable or not by a given principal, or by any honest principal, etc. A more difficult issue is handling costs over multiple

sessions. In this case, principals must not cumulate costs from correct protocol runs, but only from sessions initiated by the adversary or from malicious sessions. This requires rewriting each protocol transition in several ways, keeping track of these conditions, and tracking costs either per-session or per-principal. Modelling details are given in [11].

As an example, we discuss the Station-to-Station protocol (STS) [8] depicted in Figure 3. The protocol computes a shared session key $k = \alpha^{xy}$ starting from the random values x and y chosen by the two participants.

$$
\begin{array}{ll}
A \rightarrow B : \alpha^x & \text{A} \rightarrow \text{Adv(B)} : \alpha^x \\
B \rightarrow A : \alpha^y, Cert_B, E_k(sig_B(\alpha^y, \alpha^x)) & \text{Adv} \rightarrow \text{B} : \quad \alpha^x \\
A \rightarrow B : Cert_A, E_k(sig_A(\alpha^x, \alpha^y)) & \text{B} \rightarrow \text{Adv} : \quad \alpha^y, Cert_B, E_k(sig_B(\alpha^y, \alpha^x)) \\
 & \text{Adv(B)} \rightarrow \text{A} : \alpha^y, Cert_B, E_k(sig_B(\alpha^y, \alpha^x)) \\
 & \text{A} \rightarrow \text{Adv(B)} : Cert_A, E_k(sig_A(\alpha^x, \alpha^y))
\end{array}
$$

Fig. 3. Station to Station protocol (left) and Lowe's attack (right)

Lowe's attack [13], in the right part of Figure 3, shows the adversary capturing the message sent by A to B and resending it in his own name to B. Afterwards, B is talking to Adv, while A believes she is talking to B. Adv(B) means the adversary impersonating B, while Adv is the adversary acting as himself.

The attack found by Lowe shows a flaw in the protocol, irrespective of costs. Later, Meadows [16] analyzed this attack from a cost-based perspective. Our model allows a model checker to detect this attack automatically. By using an attack condition (`attack_state`) such as

```
dos_on_a(X, Y, P, V, L, ID) := cost(a, X).cost(i, Y).less(Y, X)
        .recv(P, a, V, L, ID) & not(send(P, a, V, L, ID))
```

we direct the model checker to find a protocol trace in which the adversary has lower cost than the honest principal A, who accepts a message from a different session. Figure 4 presents the attack trace found by CL-Atse (release 2.5-8). The attack differs slightly from the one found by Lowe, but by placing different constraints the back-end can reproduce Lowe's attack as well. The trace shows the adversary reusing a value sent by A to obtain a response from B that is further redirected and accepted by A. Steps 1 to 3 are from A's session with B (compromised by the intruder in step 3), while step 2' is from a session between the intruder and B (step 1' is implicit since the intruder knows everything sent over the network). The cost of both A and B is *high* as they compute modular exponentiations while their beliefs about the resulting shared session key (α^{xy}) are both wrong. A believes she shares a key with B, while B believes he shares the key with Adv, who is actually unable to compute it without knowing x.

The cost of the adversary is *low* as he doesn't perform computations except for redirecting messages, which is assumed to be cheap.

1. $A \to B : \alpha^x$

$$\begin{cases} \mathtt{i} \to (\mathtt{A}, 5) : \{\} \\ (\mathtt{A}, 5) \to \mathtt{i} : \mathtt{costExp}(\mathtt{g}, \mathtt{n5}(\mathtt{XA})) \\ \& \ \mathtt{Built\ from\ trans0} \end{cases}$$

2. $B \to I(A) : \alpha^{y_1}, Cert_B, E_k(sig_B(\alpha^{y_1}, \alpha^x))$

$$\begin{cases} \mathtt{i} \to (\mathtt{B}, 12) : \mathtt{costExp}(\mathtt{g}, \mathtt{n5}(\mathtt{XA})) \\ (\mathtt{B}, 12) \to \mathtt{i} : \mathtt{costExp}(\mathtt{g}, \mathtt{n6}(\mathtt{XB})).\mathtt{certB}. \\ |\mathtt{costExp}(\mathtt{g}, \mathtt{n6}(\mathtt{XB})).\mathtt{costExp}(\mathtt{g}, \mathtt{n5}(\mathtt{XA}))\_\mathtt{inv}(\mathtt{b\_pk})| \\ \_\mathtt{costExp}(\mathtt{costExp}(\mathtt{g}, \mathtt{n5}(\mathtt{XA})), \mathtt{n6}(\mathtt{XB})) \\ \& \ \mathtt{Built\ from\ trans1} \end{cases}$$

2′. $B \to I : \alpha^{y_2}, Cert_B, E_k(sig_B(\alpha^{y_2}, \alpha^x))$

$$\begin{cases} \mathtt{i} \to (\mathtt{B}, 22) : \mathtt{costExp}(\mathtt{g}, \mathtt{n5}(\mathtt{XA})) \\ (\mathtt{B}, 22) \to \mathtt{i} : \mathtt{costExp}(\mathtt{g}, \mathtt{n14}(\mathtt{XB})).\mathtt{certB}. \\ |\mathtt{costExp}(\mathtt{g}, \mathtt{n14}(\mathtt{XB})).\mathtt{costExp}(\mathtt{g}, \mathtt{n5}(\mathtt{XA}))\_\mathtt{inv}(\mathtt{b\_pk})| \\ \_\mathtt{costExp}(\mathtt{costExp}(\mathtt{g}, \mathtt{n5}(\mathtt{XA})), \mathtt{n14}(\mathtt{XB})) \\ \& \ \mathtt{Built\ from\ trans1} \end{cases}$$

3. $I(B) \to A : \alpha^{y_2}, Cert_B, E_k(sig_B(\alpha^{y_2}, \alpha^x))$

$$\begin{cases} \mathtt{i} \to (\mathtt{A}, 5) : \mathtt{costExp}(\mathtt{g}, \mathtt{n14}(\mathtt{XB})).\mathtt{certB}. \\ |\mathtt{costExp}(\mathtt{g}, \mathtt{n14}(\mathtt{XB})).\mathtt{costExp}(\mathtt{g}, \mathtt{n5}(\mathtt{XA}))\_\mathtt{inv}(\mathtt{b\_pk})| \\ \_\mathtt{costExp}(\mathtt{costExp}(\mathtt{g}, \mathtt{n5}(\mathtt{XA})), \mathtt{n14}(\mathtt{XB})) \\ (\mathtt{A}, 5) \to \mathtt{i} : \mathtt{certA}. \\ |\mathtt{costExp}(\mathtt{g}, \mathtt{n5}(\mathtt{XA})).\mathtt{costExp}(\mathtt{g}, \mathtt{n14}(\mathtt{XB}))\_\mathtt{inv}(\mathtt{a\_pk})| \\ \_\mathtt{costExp}(\mathtt{costExp}(\mathtt{g}, \mathtt{n14}(\mathtt{XB})), \mathtt{n5}(\mathtt{XA})) \\ \& \ \mathtt{Built\ from\ trans2} \end{cases}$$

Fig. 4. STS and the corresponding attack trace shown by CL-Atse

4 Combining Transitions and Horn Clauses for Detection of Guessing Attacks

A guessing attack is done by guessing the value of a low-entropy secret and being able to verify the guess. This can be done in two distinct cases. The first case requires knowing the image of a one-way function on the secret, e.g., $h(s)$, and being able to compute the function on arbitrary values for verification. The second case computes the output of a function inverse that depends on the secret (e.g., the secret is part of a decryption key), and checks a property of the output, e.g., a known part a in decrypting $\{a.m\}_s$ or two equal parts for $\{m.m\}_s$. We have presented a formalization and automation of this approach in [10].

Related work on guessing is based on various theories. A widely used definition is due to Lowe [14], while Abadi et al. [1] present a sound approach from an algebraic point of view based on indistinguishability. Corin et al. [6] also use equational theories, while [12] explicitly represents intruder computation steps, but is limited to offline attacks. Tools that are able to find guessing attacks are presented by Blanchet [5], Corin et al. [7], and Lowe [14].

4.1 Formalization of Guessing

Our approach uses oracles to represent terms obtained as functions computed over the secret. An adversary may establish two kinds of relations with an oracle: *observes* means that the adversary knows the output of the oracle, for some inputs which are not necessarily known. The stronger notion of *controls* means

that the adversary can apply inputs of its choice to the oracle and obtain the resulting output. Thus, *controls* implies *observes*, but the reverse does not hold.

With these notions defined, the first case, when the adversary knows the image of a one-way function over the secret and controls the corresponding oracle, is verified using the guessing rule:

$$observes(O^f(s)) \wedge controls(O^f(s)) \Rightarrow guess(s) \quad (1)$$

Here $O^f(s)$ denotes the oracle computing the function (term) f over secret s. Guessing holds because the adversary knows the image of a one-way function over the secret and since he controls the oracle for that function he can successively give all values of the secret as input to the oracle then verify the result. This rule can detect guessing attacks on terms such as $m.h(m, s)$ or $\{s\}_s$, etc.

In the second case, when the secret is part of the key of an invertible function we need to check if a term is verifiable by the adversary. We define a term as *verifiable* by the adversary in any of the following cases:

- the term is already known:

$$verifiable(Term) :\!- iknows(Term) \quad (2)$$

- the term is a digital signature, and the public key and message are known:

$$verifiable(apply(inv(PK), Term)) :\!- iknows(PK) \wedge iknows(Term) \quad (3)$$

- the image of a one-way function on the term and a controllable oracle for that function are known:

$$verifiable(Term) :\!- iknows(apply(F, T')) \wedge part(Term, T') \wedge controls(O^{F(T')}(Term)) \quad (4)$$

- the term is an encryption with a controllable decryption oracle, and some part T'' of the encrypted term is verifiable if the remaining part T' is added to the intruder knowledge (expressed by fact $split(Term, T', T'')$):

$$verifiable(scrypt(K, Term)) :\!- controls(O^{\{M\}_{K^{-1}}}(M)) \wedge split(Term, T', T'') \wedge verifiable(T'') \quad (5)$$

We can now define the second case of guessing. If the adversary *observes* an encryption oracle that uses a key dependent on the secret s, if he *controls* the corresponding decryption oracle (with s as input) and can *verify* a part of the encrypted message then the adversary can guess the secret, i.e.,

$$observes(O^{\{Term\}_K}(s)) \wedge controls(O^{\{Term\}_{K^{-1}}}(s)) \wedge split(Term, T', T'') \wedge verifiable(T'') \Rightarrow guess(s) \quad (6)$$

For example, consider h, m_1, $apply(h, pair(m_1, m_2))$ and $scrypt(s, m_2)$ as known. Since m_2 is a part of $apply(h, pair(m_1, m_2))$ and this function is controllable

in m_2 (h and m_1 being known), then m_2 is verifiable by rule 4. Knowing $scrypt(s, m_2)$ and controlling the decryption oracle, one can guess s.

Next, consider that $scrypt(s, scrypt(k, pair(m, m)))$ and k are known. Using rule 5, $scrypt(k, pair(m, m))$ is verifiable, since adding the first instance of m to the intruder knowledge, one can verify the second m from the pair. This in turn allows guessing s. On the other hand, from $scrypt(s, scrypt(k, m))$ and k the intruder cannot in general guess s: if symmetric encryption is done with a one-time pad then the decrypted m has no meaning and cannot be verified. Of course, if the adversary knows both m and k, the term is verifiable by rule 1 since $iknows(scrypt(k, m))$ holds. Other verification rules can be added to the model, for example in the case of authenticated encryption, etc.

To implement *observes*, the analysis performed by the model checker needs to determine if a composed term contains the secret to be guessed. We express observing an oracle as $ihears(T) \wedge contains(T, S)$, where S is the guessable secret, and *ihears* is a fact added to the right-hand side of each send transition in the protocol, supplementary to *iknows*. This new fact is needed to distinguish terms overheard on the network from terms otherwise derived by the intruder.

To decide *controls*, the analysis takes a term containing the secret and constructs a new term where the secret is replaced by a fresh value. If the adversary knows this term, it *controls* the oracle corresponding to the original term. Controlling an oracle is then modeled by the facts $replace(T, TNew) \wedge iknows(TNew)$.

For efficiency, the above expressions can be used directly in the attack condition rather than introducing explicit *observes* and *controls* facts. Moreover, a sufficient condition for the case of *offline* guessing is whether the adversary knows all parts of a term except the secret which it tries to guess. This is easier to model and verify by the back-ends and will be used in the following subsections.

We next discuss how to express the facts *contains* and *replace* for checking the presence of a secret in a term and constructing a new term, respectively, and how to efficiently model the verifiability conditions given above using Horn clauses. We evaluate the various approaches on a protocol case study.

4.2 Processing Terms That Contain the Secret

Checking whether a term contains a particular secret, and replacing it with a new value is simple conceptually but challenging in practice. These steps are the basis for detecting simple attacks such as those covered by the first guessing case of rule 1, e.g., guessing s if a and $h(a.s)$ are known. We discuss three ways of implementing containment and replacement, with major differences both in writing the rules and in the analysis time to find an attack.

We test the various approaches using as example the MS-CHAP protocol [20], depicted in Figure 1. To make analysis times more significant we modify this simple protocol by inserting more complex terms. In MS-CHAP* the last term, $H(k_{AB}, N_A)$, is changed to $H(N_A, k_{AB}, N_A)$, while for MS-CHAP** we concatenate N_A to the key seven times: $H(N_A, \ldots, N_A, k_{AB}, N_A)$. While guessing is conceptually straightforward in all cases, for complex terms the rules require more processing time, highlighting the differences between the approaches.

Naive Approach with Transitions

The straightforward approach is to consider terms heard by the intruder on the network, to define *contains* and *replace* for atoms, and to derive them for composed terms using the corresponding facts for their components. For atomic terms, rules can be defined as shown in the left part of Figure 5 for an atom of type *text*. Since the atom is not equal to the secret, *replace* does not change it, and *contains* is asserted with a dummy null secret.

For composed terms, *contains* and *replace* facts can be derived if these facts are known for their components. This is shown in the right part of Figure 5 for a term composed with *pair*, and similar rules are defined for terms composed with *scrypt*, *crypt*, etc. In our implementation, we restrict the application of these rules only for terms sent over the network, in order to avoid their inefficient application over the entire intruder knowledge. This is because the set of terms known by the intruder is large, conceptually even unbounded (since the intruder can always create fresh terms or compose already known terms with known operators), and the back-end may fail to terminate. Using *ihears* instead of *iknows* also allows us to derive the sub-terms of terms that are heard on the network, which can be easily done by adding transitions for each type of composed term.

In practice, this modelling approach works for decomposing simple terms, however if the terms are more complex, and many transitions are needed, then the back-end times out attempting to verify the model. As seen in Table 1, this modelling variant fails for terms on which the forthcoming procedures succeed.

```
state_process(A, ID, 0)
.ihears(AtomText)
.not(equal(AtomText, s))
=>
state_process(A, ID, 0)
.ihears(AtomText).
.contains(AtomText, snull)
.replace(AtomText, AtomText)
```

```
state_process(A, ID, 0)
.ihears(pair(T1, T2))
.contains(T1, S1).contains(T2, S2)
.replace(T1, T1New).replace(T2, T2New)
=>
state_process(A, ID, 0)
.ihears(pair(T1, T2))
.contains(T1, S1).contains(T2, S2)
.replace(T1, T1New).replace(T2, T2New)
.contains(pair(T1, T2), S1)
.contains(pair(T1, T2), S2)
.replace(pair(T1, T2), pair(T1New, T2New))
```

Fig. 5. Contains and replace for atomic terms (left) and composed terms (right)

Improved Approach with Transitions

The previous approach is inefficient because the steps for customized intruder deductions can be interleaved with protocol steps, leading to exponential complexity. To avoid this, we control the order in which the terms are processed by placing them in a stack (constructed with *pair* and a dummy separator). Terms are processed by structural decomposition and each new sub-term is placed on top of the stack unless it is an atom and the *contains* and *replace* facts for it can

be directly deduced. Clearly an atom contains the secret if and only if it is the actual secret, otherwise *contains* is false and *replace* leaves the atom unchanged.

For example, consider the term $scrypt(k, h(pair(N_A, N_B)))$ heard over the network. In the first step the stack contains only this term. Next, k (the left operand of *scrypt*) is added to the top of the stack. As this operand is atomic, one can directly establish *contains* and *replace* for it, and remove it from the stack. The next item on the stack will be the right operand of *scrypt*, i.e., $h(pair(N_A, N_B))$. The next element is $pair(N_A, N_B)$, with the stack now containing three items, and so on. On the left side of Figure 6 an atom of type *text* is eliminated from the list, while on the right side a composed term is split into its components. This mechanism greatly reduces complexity due to interleaving.

As can be seen in Table 1, this modelling variant succeeds in deriving the guess also for the artificially complicated structure of MS-CHAP**. However, its time requirement increases significantly with the complexity of the term, a drawback removed in the next modelling solution.

```
state_process(A, ID, 1)
.process(
 pair(pair(AtomText,sep),Right))
.not(guessable(AtomText))
=>
state_process(A, ID, 0)
.process(Right)
.contains(AtomText, snull).
.checked(AtomText)
.replace(AtomText, AtomText).
.replaced(AtomText)
```

```
state_process(A, ID, 1)
.process(
 pair(pair(pair(T1, T2), sep), Right))
.contains(T1, S1).replace(T1, T1New)
.contains(T2, S2).replace(T2, T2New)
=>
state_process(A, ID, 0)
.process(Right)
.contains(T1, S1).replace(T1, T1New)
.contains(T2, S2).replace(T2, T2New)
.checked(pair(T1, T2))
.contains(pair(T1, T2), pair(S1, S2))
.replace(pair(T1,T2),pair(T1New,T2New))
.replaced(pair(T1, T2)).
```

Fig. 6. Improved contains/replace for atomic terms (left) and composed terms (right)

Efficient Approach with Horn Clauses

Horn clauses are more elegant and intuitive for modelling intruder deductions. They were specially introduced in ASLan for this purpose, as well as for modelling static or dynamic security policies.

The model checkers of the AVANTSSAR platform process Horn clauses in different ways, depending on their overall exploration strategy. CL-Atse employs a backward search, using Horn clauses only when deriving some fact is required (e.g., for the left-hand side of a transition). SATMC on the other hand employs a forward strategy and saturates the set of known facts by transitively applying Horn clauses after each transitions. Thus, Horn clauses written with one search strategy in mind may lead a model checker employing the opposite strategy to non-termination. We have devised models adapted to the use of CL-Atse.

For example, the Horn clauses in Figure 7 find both the part of a term and its remainder. The fact $ispart(T_1, T_2, T_3)$ denotes that T_1 is split into disjoint parts T_2 and T_3. The Horn clause *part_left* states that T_2 is part of $pair(T_0, T_1)$ with remainder $pair(T_0, T_3)$ if T_2 is part of T_1 with remainder T_3. Such rules need to be written for all operators that can be applied on terms.

```
hc part_null(T1) :=
        ispart(T1, null, T1)
hc part_id(T1) :=
        ispart(T1, T1, null)
hc part_left(T0, T1, T2, T3) :=
        ispart(pair(T0, T1), T2, pair(T0, T3)) :- ispart(T1, T2, T3)
hc part_right(T0, T1, T2, T3) :=
        ispart(pair(T0, T1), T2, pair(T3, T1)) :- ispart(T0, T2, T3)
hc part_scrypt_left(T0, T1, T2, T3) :=
        ispart(scrypt(T0, T1), T2, pair(T0, T3)) :- ispart(T1, T2, T3)
hc part_scrypt_right(T0, T1, T2, T3) :=
        ispart(scrypt(T0, T1), T2, pair(T3, T1)) :- ispart(T0, T2, T3)
```

Fig. 7. Splitting terms using Horn clauses

1. $A \rightarrow B$: A — $\begin{cases} \mathtt{i} \rightarrow (\mathtt{chap\_Init}, 11) : \mathtt{start} \\ (\mathtt{chap\_Init}, 11) \rightarrow \mathtt{i} : \mathtt{a} \end{cases}$

2. $B \rightarrow A$: N_B — $\begin{cases} \mathtt{i} \rightarrow (\mathtt{chap\_Resp}, 18) : \mathtt{a} \\ (\mathtt{chap\_Resp}, 18) \rightarrow \mathtt{i} : \mathtt{n4(Nb)} \end{cases}$

3. $A \rightarrow B$: N_A, $H(k_{AB}, N_A, N_B, A)$ — $\begin{cases} \mathtt{i} \rightarrow (\mathtt{chap\_Init}, 13) : \mathtt{n4(Nb)} \\ (\mathtt{chap\_Init}, 13) \rightarrow \mathtt{i} : \mathtt{pair(n2(Na), h(pair(s, pair(n2(Na), pair(Nb(2), a)))))} \end{cases}$

4. $B \rightarrow A$: $H(k_{AB}, N_A)$ — $\begin{cases} \mathtt{i} \rightarrow (\mathtt{chap\_Resp}, 20) : \mathtt{pair(n2(Na), h(pair(s, pair(n2(Na), pair(n4(Nb), a)))))} \\ (\mathtt{chap\_Resp}, 20) \rightarrow \mathtt{i} : \mathtt{h(pair(s, n2(Na)))} \end{cases}$

Horn clause facts:

```
controls(h(pair(s,n2(Na))),s),
iguess(s),
ihears(h(pair(s,n2(Na)))),
ispart(h(pair(s,n2(Na))),h(pair(s,n2(Na))),null),
ispart(s,pair(s,n2(Na)),pair(n2(Na),null)),
ispart(s,s,null),
observes(h(pair(s,n2(Na))),s)
```

Fig. 8. MS-CHAP v2 and the corresponding attack trace found by CL-Atse

In the attack trace from Figure 8 the intruder was forced to guess k_{AB} from the message in step 4, although it could have also guessed at step 3. The Horn clauses show that the intruder *observes* the term from step 4, and repeatedly applies rules involving *ispart* until it can derives *controls*, which then allows the guess. Rules for *observes* and *controls* are discussed in the next subsection.

Table 1 shows that by using this approach the increase in time requirements is negligible for more complex terms which otherwise require several seconds of processing, or even fail if naive transition rules are used for term processing.

Table 1. Timing results for attack detection on MS-CHAP with CL-Atse

	MS-CHAP	MS-CHAP*	MS-CHAP**
Naive Transitions	456 ms	820 ms	TOUT
Efficient Transitions	1272 ms	1812 ms	10529 ms
Horn Clauses	120 ms	120 ms	112 ms

4.3 Using Horn Clauses and Transitions for Intruder Deductions

Finally, to flag a guessing attack, we need to determine whether some term is verifiable by the intruder. Figure 9 shows rules for the verifiability conditions discussed previously. To achieve this, in some cases we need to add terms to the intruder knowledge as shown in Figure 10. This is important for modelling: while Horn clauses can be used for verifying terms, they cannot be used to add terms to intruder knowledge when working with CL-Atse, due to the backward strategy it employs when using Horn clauses. (SATMC however can do this, as it due to the forward strategy employed). The two guessing cases are detected by the Horn clauses in Figure 11. Thus, to validate the guess we have to use a mixture of Horn clauses and intruder transitions. Using these, CL-Atse is able to detect guessing from terms such as $\{m, m\}_s$ or $k, \{\{m, m\}_k\}_s$, etc.

```
% verify known term
hc verif_iknows(MsgA) :=
    verifiable(MsgA) :- iknows(MsgA)

% verify signature
hc verif_sign(PbK, MsgA) :=
    verifiable(apply(inv(PbK), MsgA)) :- iknows(PbK), iknows(MsgA)

% verification of term under hash
hc verif_hash(MsgA, MsgB, MsgC) :=
    verifiable(MsgA) :- iknows(apply(h,MsgB)),
                        ispart(apply(h,MsgB), MsgA, MsgC), iknows(MsgC)

% the ciphertext is verifiable if the encryption key is known
% and part of the plaintext is verifiable
hc verif_scrypt_ciphertext(K, MsgA, MsgB, MsgC) :=
    verifiable(scrypt(K, MsgA)) :- iknows(K), split(MsgA, MsgB, MsgC),
                                   verifiable(MsgC)
```

Fig. 9. Horn clauses for verifying terms

```
% split a message if it was not split before
step trans_split(A, MsgA, MsgB, MsgC, K):=
  state_split(A)
  .ihears(scrypt(K, MsgB)).ispart(MsgB, MsgA, MsgC)
  .not(equal(MsgC, null)).not(is_split(MsgB))
  =>
  state_split(A)
  .ihears(scrypt(K, MsgB)).ispart(MsgB, MsgA, MsgC)
  .iknows(MsgA)
  .split(MsgB, MsgA, MsgC).is_split(MsgB)
```

Fig. 10. Transition for adding terms to intruder knowledge

```
hc controls_hash(S, K, Rest, Msg) :=
controls(apply(h, Msg), S) :- ihears(apply(h, Msg)),
                              ispart(S, Msg, Rest), iknows(Rest)

hc observes_hash(S, K, Rest, Msg) :=
observes(apply(h, Msg), S) :- ihears(apply(h, Msg)),
                              ispart(S, Msg, Rest)

hc guess_case_i(S, Msg) :=
iguess(S) :- lowentropy(S), observes(Msg, S), controls(Msg, S)

hc controls_scrypt(S, K, KRest, Msg) :=
controls(scrypt(K, Msg), S) :- ihears(scrypt(K, Msg)),
                               ispart(S, K, KRest), iknows(KRest)

hc observes_scrypt(S, K, KRest, Msg) :=
observes(scrypt(K, Msg), S) :- ihears(scrypt(K, Msg)),
                               ispart(S, K, KRest)

hc guess_case_ii(S, K, MsgA, MsgB, MsgC) :=
iguess(S) :- lowentropy(S),
             observes(scrypt(K, MsgA), S), controls(scrypt(K, MsgA), S),
             split(MsgA, MsgB, MsgC), verifiable(MsgC)
```

Fig. 11. Horn clauses for guessing

4.4 Distinguishing Detectable from Undetectable On-line Attacks

With the guessing mechanism established above, the attack condition can be stated in different flavours. For example, as the deduction rules allow detecting on-line attacks, we can ask whether the attack is detectable or not by some (or any) honest participant. The relevance of this kind of undetectable on-line attacks was previously pointed out by Ding and Horster [9].

We can express that guessing is undetectable for honest participants if for all executions where guessing happens, the protocol is completed normally by all participants. Thus, we can reformulate undetectable guessing as a reachability

check for an attack state in which the secret has been guessed *and* all protocol participants have completed execution.

In ASLan models, each participant has a unique identifier `ID` which is part of its `state` fact. We also define for each participant the fact `running(ID)` which is true in every state except the participant's initial and final states. An adversary *observes* (*controls*) an oracle *undetectably* if it *observes* (*controls*) the oracle and all protocol participants reach a final state, i.e., no fact `running(ID)` holds.

A protocol description can be automatically augmented to allow for this check by statically identifying its *initial* and *final* transitions. Initial transitions have in the LHS a *state* fact, whereas *final transitions* have in the RHS a *state* fact that does not appear in the LHS of any other transition rule. Every initial transition is augmented to generate a fresh `ID` value, and the positive fact `running(ID)` is added to its RHS. Every *final transition* is augmented with the fact `running(ID)` on the LHS, but not on the RHS, thus it becomes false. This protocol adaptation allows to directly express undetectable guessing.

The same technique can be used to distinguish offline attacks. This is achieved by checking for attacks, while requiring that no fresh ID is ever generated. It may also be useful to check, for instance, if only adversary *observe* actions were done on-line, while *controls*, which involves computations and is more tedious, is performed offline. This can be done by checking that no fresh ID is generated between *observes* and *controls*. Thus, our approach allows not only the detection of guessing attacks, but also their classification.

5 Conclusions

As model checkers for security protocols do not by default support the detection of all attacks, one needs to use customized intruder deductions and transitions for this purpose. This allows the handling of new types of attacks without changing the model-checking back-ends. In this paper, we have explored two such case studies: modelling guessing attacks and denial of service by resource exhaustion. These attacks are relevant as many protocols used in practice are vulnerable to them, and we show the applicability of our theories with automatically obtained attack traces on known protocols.

We present different modelling options and investigate the relative efficiency of transition rules and Horn clauses, with the latter providing significant performance gain and allowing the processing of more complex message terms. The modelling approaches described here show the power of the ASLan specification language which serves as input to the AVANTSSAR model checkers.

We hope that the approaches shown here can provide a starting point for modelling other types of attacks that are currently not detected by default.

References

1. Abadi, M., Baudet, M., Warinschi, B.: Guessing attacks and the computational soundness of static equivalence. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 398–412. Springer, Heidelberg (2006)

2. Armando, A., Compagna, L.: SAT-based model-checking for security protocols analysis. International Journal of Information Security 7(1), 3–32 (2008)
3. AVANTSSAR: Deliverable 2.3 (update): ASLan++ specification and tutorial (2011), http://www.avantssar.eu
4. Basin, D.A., Mödersheim, S., Viganò, L.: OFMC: A symbolic model checker for security protocols. Internat. J. of Information Security 4(3), 181–208 (2005)
5. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: Proc. 14th IEEE Computer Security Foundations Workshop, pp. 82–96 (2001)
6. Corin, R., Doumen, J.M., Etalle, S.: Analysing password protocol security against off-line dictionary attacks. In: Proc. 2nd Int'l. Workshop on Security Issues with Petri Nets and other Computational Models (WISP), pp. 47–63 (2004)
7. Corin, R., Malladi, S., Alves-Foss, J., Etalle, S.: Guess what? Here is a new tool that finds some new guessing attacks. In: Proc. Workshop on Issues in the Theory of Security, pp. 62–71 (2003)
8. Diffie, W., van Oorschot, P.C., Wiener, M.J.: Authentication and authenticated key exchanges. Designs, Codes and Cryptography 2(2), 107–125 (1992)
9. Ding, Y., Horster, P.: Undetectable on-line password guessing attacks. Operating Systems Review 29(4), 77–86 (1995)
10. Groza, B., Minea, M.: A formal approach for automated reasoning about off-line and undetectable on-line guessing (short paper). In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 391–399. Springer, Heidelberg (2010)
11. Groza, B., Minea, M.: Formal modelling and automatic detection of resource exhaustion attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS (2011)
12. Hankes Drielsma, P., Mödersheim, S., Viganò, L.: A formalization of off-line guessing for security protocol analysis. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 363–379. Springer, Heidelberg (2005)
13. Lowe, G.: Some new attacks upon security protocols. In: Proc. of the 9th IEEE Computer Security Foundations Workshop, pp. 162–169 (1996)
14. Lowe, G.: Analysing protocols subject to guessing attacks. Journal of Computer Security 12(1), 83–98 (2004)
15. Matsuura, K., Imai, H.: Modification of internet key exchange resistant against denial-of-service. In: Pre-Proceedings of Internet Workshop, pp. 167–174 (2000)
16. Meadows, C.: A cost-based framework for analysis of denial of service networks. Journal of Computer Security 9(1/2), 143–164 (2001)
17. Ramachandran, V.: Analyzing DoS-resistance of protocols using a cost-based framework. Tech. Rep. DCS/TR-1239, Yale University (2002)
18. Smith, J., González Nieto, J.M., Boyd, C.: Modelling denial of service attacks on JFK with Meadows's cost-based framework. In: Proc. of the 4th Australasian Information Security Workshop, pp. 125–134 (2006)
19. Turuani, M.: The CL-Atse protocol analyser. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 277–286. Springer, Heidelberg (2006)
20. Zorn, G.: Microsoft PPP CHAP extensions, version 2 (2000)