An Improved Constraint-Based System for the Verification of Security Protocols

Ricardo Corin[1] and Sandro Etalle[1,2]

[1] Faculty of Computer Science,
University of Twente, P.O.Box 217, 7500AE Enschede, The Netherlands
[2] CWI, Center for Mathematics and Computer Science Amsterdam
{corin,etalle}@cs.utwente.nl

Abstract. We propose a constraint-based system for the verification of security protocols that improves upon the one developed by Millen and Shmatikov [30]. Our system features (1) a significantly more efficient implementation, (2) a monotonic behavior, which also allows to detect flaws associated to partial runs and (3) a more expressive syntax, in which a principal may also perform explicit checks. In this paper we also show why these improvements yield a more effective and practical system.

1 Introduction

Cryptographic protocols are the essential means for the exchange of confidential information and for authentication. Their correctness and robustness are crucial for guaranteeing that a hostile intruder can not get hold of secret information (e.g. a private key) or to force unjust authentication. Unfortunately, the design of cryptographic protocols appears to be rather error-prone. A great deal of published protocols has later been shown to contain errors prejudicing their safety. This stimulated research on formal verification of security protocols see e.g. [19, 10, 28, 31, 27, 35, 36, 32, 17, 18]. In this setting several approaches are based on Dolev and Yao's [19], where it is proposed to test a protocol explicitly against a hostile intruder who has complete control over the network, and who can intercept and forge messages. By an exhaustive search, one can establish whether the protocol is flawed or not [31, 29, 12, 23]. A crucial aspect in this approach is try to limit the search space explosion that occurs when modeling the intruder's behavior.

In this respect, an elegant and effective solution was recently proposed by Millen and Shmatikov [30] (we will refer to their system as MS in the future). Security properties, such as secrecy and authentication, can be characterized as reachability problems; in MS it is shown how to convert a reachability problem into a constraint solving problem. They use the strand space model (originally developed by Thayer, Herzog and Guttman [21]) for honest processes, and a term set closure characterization for the attacker. One limitation of this approach is that the number of sessions of the protocol must be bounded. This is the price one must pay for an automatic full decision procedure.

At the University of Twente, we have extensively used MS for the analysis of security and non-repudiation protocols. The system we present here is an improvement and extension of MS, dictated by our practical experience. In particular, this system improves on MS thanks to the following features:

- **A *monotonic* behavior** As part of the specification, a system scenario must be declared, stating which protocol instances should be present in the system to be checked. This scenario, called *semibundle*, will be defined more precisely in section 2. With *monotonocity* we indicate the fact that, the system analyses not only the traces for the original semibundle but also possible solutions for *smaller* semibundles. Dually, flaws exhibited by a semibundle are exhibited by larger semibundles as well. This is important in practice. For instance, it allows to easily detect flaws associated to incomplete runs, like the one shown by Lowe [26] on the Woo and Lam mutual authentication protocol [40].
- **More expressiveness** In particular, it allows the principals to perform explicit checks. Security protocols may perform tests at some stage of their execution. To model this in a natural yet accurate way, it is necessary to extend the definition of strands. We add a new special operation called the *check* operation. This action can be performed by any honest principal when she wants to perform a test for deciding whether to continue or not her protocol execution. This newly added operation improves the expressiveness of the system and allows the user to verify a broader class of protocols.
- **A more efficient semantics.** We modified the semantics of the system to make it simpler and more intuitive. As it turns out, this change leads to a significantly more efficient execution.

This paper is organized as follows. Section 2 contains the preliminary definitions and the description of the new semantics. In Section 3 we show how *explicit checks* work, and why they are necessary. In Section 4 we show that the system exhibits a monotonic behavior, and why this has practical advantages. Furthermore, in Section 5 we report some benchmarks. Conclusions are drawn in Section 6 and finally we conclude in Section 7 mentioning some related work.

2 Constraint-Based Protocol Analysis

2.1 Preliminaries

The reader is assumed to be familiar with the terminology and the basic results of the semantics of logic programs [4, 24]. In particular, we assume familiarity with the concept of (most general) unifier. Moreover, we use Prolog notation for distinguishing variables (that start with an uppercase letter, or an underscore) from non-variable terms. The only exception to this rule is that we follow the standard security notation in which names of principals are denoted by uppercase letters (A, B, I).

Parametric strands We now introduce in an informal way the basic notions used in the sequel. For a more formal approach we refer to [30, 21, 39]. Given a term t, $+t$ and $-t$ are *events*, in particular $+t$ is a *send* event, and $-t$ is a *receive* event; in both cases, t is the contents of the message. Messages are built according to the free algebra generated by the following operators (we show in parenthesis the notation adopted in the implementation):

- $[t1, t2]$ (`[t1,t2]`) pairing.
- $pk(P)$ (`pk(P)`) public key of P.
- $h(t)$ (`sha(t)`) the hash of t.
- $[t]_k^{\leftrightarrow}$ (`t+k`) term t encrypted with k using a symmetric key algorithm.
- $[t]_k^{\rightarrow}$ (`t*k`) term t encrypted with k using a public key algorithm.
- $sig_k(t)$ (`t/k`) public key signature of t that is validated using key k. We assume that private keys of a key pair are never leaked, thus the attacker can only construct its own signatures.
- ε (`e`). Constant representing the identity of the attacker.

A sequence of events is called a (parametrized, as it might contain variables) *strand*. A strand can represent thus the role of a principal in a protocol, and is mostly used so. For instance, the initiator role of the Needham-Schroeder-Lowe public-key protocol handshake (NSL) [25] is specified as:

$$Init(A, B, N_A, N_B) = +[A, N_A]_{pk(B)}^{\rightarrow} - [N_A, N_B, B]_{pk(A)}^{\rightarrow} + [N_B]_{pk(B)}^{\rightarrow}$$

The responder role $Resp(A, B, N_A, N_B)$ is the same except that $+$ and $-$ are interchanged.

A *semibundle* is a (finite) set of parametrized strands. Thus, for example,

$$\{Init(a, B, n_a, N_B)\}, Resp(A, b, N_A, n_b)\}$$

is a semibundle. Notice that we instantiated A with a and N_A with n_a in $Init$, as well as B with b and N_B with n_b in $Resp$. In the first case, this specifies that we want the honest principal a to take the initiator role, and to introduce the fresh nonce[1] n_a (and similarly for the $Resp$ role). The communication model is that of Dolev and Yao, in which the (malicious) intruder has complete control over the network. Actually the intruder is identified with the network itself: when a principal sends a message, it actually passes it to the intruder; on the other hand, when a principal needs to receive a message, the system checks if the intruder is able to generate a message of the appropriate format.

2.2 A New Semantics

To explain the semantics of the system, we employ a more intuitive (and algorithmic) approach than MS and define the following notion of a (partial) *run*. Given a semibundle S and an intruder's initial knowledge K, a (partial) *run* is a reiterated sequence of the following steps:

---

[1] A nonce is a randomly generated value, usually employed to defeat replay attacks.

1. a non-empty strand in S is chosen nondeterministically, and its first event e is removed from it,
2. if the event is a send, $e = +t$, then t is added to the knowledge of the intruder, $K := K \cup \{t\}$
3. if the event is a receive, $e = -t$, then it is checked if the attacker ε can generate (using the knowledge K) an instance t' of t. In that case, if $\theta = mgu(t, t')$ then the remaining semibundle S and the intruders knowledge are instantiated by θ (e.g., $S := S\theta$, $K := K\theta$), and the computation proceeds from (1). If ε cannot generate such a message, then the run stops.

To avoid an infinitely branching system, we have to pay attention at step (3), in fact the intruder might be able to generate an infinite number of instances of t. To solve this problem, the constraint solving system works in the following *lazy* fashion: t' is constructed incrementally (by incremental instantiation). As soon as t' becomes an instance of t then the construction process is suspended, (the mgu θ is applied to K and S automatically) and the system proceeds with step (1). The suspended construction process is later resumed in the case that some subsequent action (e.g., a check) instantiates t in such a way that t' is not any longer an instance of it. A resumed construction process yields either to a new (more instantiated) mgu θ or to failure.

The knowledge of the intruder contains initially only the names of the agents involved in the protocol. Clearly, a run contains always a finite number of steps. In MS it is demonstrated that the process of checking whether ε can generate a message t always terminates. Therefore one can generate all runs in a finite time and full nondeterminism is achieved by means of backtracking. We say that the run is *complete* if after the last step the semibundle contains only empty strands, otherwise we call it a *partial run*. (In the notation of MS, a complete run is called a *bundle*).

While we adopted this semantics basically for clarity reasons, it turned out to lead to a significantly faster implementation than MS. This will be explained in Section 5.

Finally, we need the following order relationship $\preceq$.

Definition 1. *Let s and s' be strands and S, S' be semibundles. We say that*

- $s \preceq s'$, *if s is a prefix of s', and that*
- $S \preceq S'$, *If there exists a variable renaming θ for which, for each $s \in S\theta$ there exists a distinct $s' \in S'$ such that $s \preceq s'$.*

3 Explicit Checks

Our first extension consists of the addition of a new event to the strand space, the *check event*. To motivate its necessity, consider the following protocol (inspired by Zhou-Gollman [41]):

Message 1. $A \rightarrow B : [L, C]$
Message 2. $B \rightarrow A : sig_{pk(B)}(C)$
Message 3. $A \rightarrow B : [sig_{pk(A)}(K), K]$
Message 4. $B \rightarrow A : sig_{pk(B)}([L, C, K])$

Here, if K is a symmetric key and M a message, $C = [M]_K^{\leftrightarrow}$ is obtained by symmetric encryption of M by K, and finally $L = h(M)$ is obtained by hashing M. In message 1, A starts the session by sending to B the hash L and a ciphered message C. B replies with his signature of C. In message 3, A sends the signed symmetric key K to B (we need to send K also in plain since our signature operator is not invertible). Now, *and not before*, B can decrypt C and check that $L = h(M)$. If the check does not succeed then B stops the session, otherwise he continues by sending message 4.

A Vulnerability The above protocol presents the following vulnerability: the attacker can impersonate a honest principal (A, the initiator), and convince B that he is communicating with A. The attack comes to light by instantiating the protocol twice (sessions α and β) as follows:

Message α.1. $I_A \rightarrow B : [h(M), [M]_K^{\leftrightarrow}]$
Message α.2. $B \rightarrow I_A : sig_{pk(B)}([M]_K^{\leftrightarrow})$
Message β.1. $I \rightarrow A : [*, K]$
Message β.2. $A \rightarrow I : sig_{pk(A)}(K)$
Message α.3. $I_A \rightarrow B : [sig_{pk(A)}(K), K]$
Message α.4. $B \rightarrow I_A : sig_{pk(B)}([h(M), [M]_K^{\leftrightarrow}, K])$

First the intruder starts to impersonate A in message α.1 (I_A is the standard notation for indicating the intruder I impersonating the honest principal A). He sends $L = h(M)$ and $C = [M]_K^{\leftrightarrow}$, for some M and K. B then replies signing of $[M]_K^{\leftrightarrow}$. Now the intruder must provide the message $sig_{pk(A)}(K)$. For this, he initiates a new session β with A, and composes a new message with anything as L, represented as $*$ (it does not matter what value it takes) but with K as C (message β.1). Then A replies by signing K, and this is exactly what the intruder needs (message β.2). He just creates a new message concatenating this message and K, and sends it to B (message α.3). B receives the message, performs successfully the check and finishes his session by sending the last message α.4. Note also that the β session is partial.

3.1 Extended Strands

In order to implement the protocol correctly, we need to introduce a new event to the strand model, the *check*. A check event has the syntax $check(t_1 = t_2)$, where t_1 and t_2 are terms of the free algebra defined in Section 2. We call *extended* strand (resp. bundles) a parametric strand (bundle) in which check events are allowed. The definition of a *run* is extended in the straightforward way to include check: when a check event is selected, if t_1 and t_2 can be reduced to the same

(most general) term t, so that $t_1\theta = t_2\theta = t$, then the remaining semibundle and the intruder's knowledge are instantiated by θ, and the run proceeds, otherwise the run stops (in this latter case, we have a partial run).

Using extended strands, the above protocol can be specified in Prolog as follows (here we only show the specification of the responder role B, which is sufficient for finding the attack):

```
responder(A, B, L, C, K, M, [recv([L, C]),
                            send(C/pk(B)),
                            recv([K/pk(A),K]),
                            check(C=M+K),
                            check(L=sha(M)),
                            send([L, [C, K]]/pk(B)]).
```

To find the above attack we need to have at least two parallel sessions, actually we need a semibundle with at least two responder roles. In Prolog, the semibundle specification is the following.

```
semibundle([Sa,Sb]):- responder(a,b, L, C1, k, m, Sa),
                      responder(A,B, L, C2, k, m, Sb).
```

Now, what we have to check is whether there exists a run in which the first strand can complete its role. The output we obtained from the system corresponds with the attack described previously:

```
trace:
[responder(a, b, sha(m), m + k, k, m), recv([sha(m), m + k])]
[responder(a, b, sha(m), m + k, k, m), send((m + k) / pk(b))]
[responder(e, a, sha(m), k, k, m), recv([sha(m), k])]
[responder(e, a, sha(m), k, k, m), send(k / pk(a))]
[responder(a, b, sha(m), m + k, k, m), recv([k / pk(a), k])]
[responder(e, a, sha(m), k, k, m), recv([k / pk(e), k])]
[responder(a, b, sha(m), m + k, k, m), check(m + k = m + k)]
[responder(a, b, sha(m), m + k, k, m), check(sha(m) = sha(m))]
[responder(a, b, sha(m), m + k, k, m), send([k, [m + k, sha(m)]] / pk(b))]
reached state:
[[responder(a, b, sha(m), m + k, k, m), []], [responder(e, a, sha(m), k, k,
m), [check(k = m + k), check(sha(m) = sha(m)), send([k, [k, sha(m)]] /
pk(a))]]]
```

The trace reported by the system is a list of pairs of the form `[role(X),
event(M)]`; this should be read as "*role X* performs *event* with message M". Exploring the output, we can notice the instantiations of A and B to ε and a respectively given by the system.

3.2 Pattern Matching Approach

We now go back to the original system MS to show that, in that context, the above attack cannot be found in a natural way. Actually, there are two reasons

why this is so: the first one is because MS does not allow explicit checks, and the second (also important) reason is that in MS only complete runs are considered, while the above attack comes from a *partial run* (this problem is addressed in Section 4).

Without having explicit checks, the natural way of performing the check after message 3 is by directly substituting in the first line L with $h(M)$ and C with $[M]_K^{\leftrightarrow}$ and modify the rest of the protocol accordingly.

Message 1. $A \rightarrow B : [h(M), [M]_K^{\leftrightarrow}]$
Message 2. $B \rightarrow A : sig_{pk(B)}([M]_K^{\leftrightarrow})$
Message 3. $A \rightarrow B : [sig_{pk(A)}(K), K]$
Message 4. $B \rightarrow A : sig_{pk(B)}([h(M), [M]_K^{\leftrightarrow}, K])$

This protocol seems equivalent to the original one, but it is not so: here B can find out *already at step 1* if the first message does not comply with the definition, while in practice (and in the original definition) this can happen only after step 3. The proof of this is the fact that the modified protocol is *not flawed*: the above attack does not apply to it.

Concluding this section, checks are a simple yet crucial extension to the strand space models. The above protocol shows a situation where if checks are not used an attack might not be found.

It is worth mentioning that the protocol we reported here is not contrived, but it is actually similar to a subprotocol of the Zhou-Gollmann optimistic non-repudiation protocol [41] we have been working on for some time, where the same kind of check for testing the validity of M is carried out. In fact, the necessity of explicit checks became clear to us when we started using strand spaces for analyzing non-repudiation protocols. We don't report here the full [41] because non-repudiation protocols are outside the scope of this paper.

4 Monotonicity and Partial Runs

Let us fix some (not-too-formal) notation.

We say that a protocol has an (elementary) *authentication flaw* if there is a run in which one of the principals (e.g., the responder) at some points receives a message t while the other principal (e.g., the initiator) has not previously sent message t (this definition applies to the case of one session, for more sessions the extension is straightforward).

In addition, we say that a protocol presents a *secrecy flaw* if at some point a message that was to be kept secret can be known by the intruder. Both authentication and secrecy flaws can be checked with our tool; in particular, to check secrecy one can add the singleton strand $[-t]$ to the semibundle, where t is the message that should remain secret. If there is a run in which $-t$ is selected and "resolved", then the protocol has a flaw.

Thanks to its semantics, our system enjoys the following important monotonicity property.

Proposition 2 (Monotonicity). *Let S and S' be two semibundles such that $S \preceq S'$. Any authentication or secrecy flaw exhibited by S is exhibited by S' as well.*

This is of crucial practical importance. For instance, some vulnerabilities may only be found when considering partial runs. This is the case of the Woo and Lam mutual authentication protocol [40], which we introduce next. A possible attack of this protocol is described by Lowe [26]. The protocol aims at establishing a session key and provides mutual authentication between two agents A and B, with the help of a trusted server S.

Message 1. $A \rightarrow B : [A, N_A]$
Message 2. $B \rightarrow A : [B, N_B]$
Message 3. $A \rightarrow B : [A, B, N_A, N_B]^{\leftrightarrow}_{K_{AS}}$
Message 4. $B \rightarrow S : [A, B, N_A, N_B]^{\leftrightarrow}_{K_{AS}}, [A, B, N_A, N_B]^{\leftrightarrow}_{K_{BS}}$
Message 5. $S \rightarrow B : [B, N_A, N_B, K_{AB}]^{\leftrightarrow}_{K_{AS}}, [A, N_A, N_B, K_{AB}]^{\leftrightarrow}_{K_{BS}}$
Message 6. $B \rightarrow A : [B, N_A, N_B, K_{AB}]^{\leftrightarrow}_{K_{AS}}, [N_A, N_B]^{\leftrightarrow}_{K_{AB}}$
Message 7. $A \rightarrow B : [N_B]^{\leftrightarrow}_{K_{AB}}$

First A sends a nonce to B (message 1), and then B responds by returning another nonce (message 2). A then sends to B an encrypted message for the server (message 3). B forwards this message to the server, along with a similar message of his own (message 4).
When the server receives this message, he checks that the agent identities and nonces match. If so, the server selects a new key K_{AB} to be used in the subsequent session between A and B. Subsequently, the server creates two encrypted components, one for each of A and B, and returns both of them to B (message 5). B decrypts his component to obtain the key, and then forwards the other component to A, along with a component encrypted with K_{AB} (message 6). Finally, A decrypts his component to learn the key, and returns a message to B (message 7).

The Attack The attack spans two sessions, α and β:

Message $\alpha.1$. $I_A \rightarrow B : [A, B]$
Message $\alpha.2$. $B \rightarrow I_A : [B, N_A]$
Message $\alpha.3$. $I_A \rightarrow B :$ bit-string
Message $\alpha.4$. $B \rightarrow I_S :$ bit-string, $[A, B, B, N_B]^{\leftrightarrow}_{K_{BS}}$
Message $\beta.1$. $I_A \rightarrow B : [A, N_B]$
Message $\beta.2$. $B \rightarrow I_A : [B, N'_B]$
Message $\beta.3$. $I_A \rightarrow B :$ bit-string$'$
Message $\beta.4$. $B \rightarrow I_S :$ bit-string$'$, $[A, B, N_B, N'_B]^{\leftrightarrow}_{K_{BS}}$
Message $\alpha.5$. $I_S \rightarrow B :$ bit-string$''$, $[A, B, N_B, N'_B]^{\leftrightarrow}_{K_{BS}}$
Message $\alpha.6$. $B \rightarrow I_A :$ bit-string$''$, $[B, N_B]^{\leftrightarrow}_{N'_B}$
Message $\alpha.7$. $I_A \rightarrow B : [N_B]^{\leftrightarrow}_{N'_B}$

In message $\alpha.1$, the intruder starts to impersonate A to attack B, *choosing B as the nonce*. B responds in message $\alpha.2$ by returning a nonce N_B. Then, the intruder sends any bit-string of the appropriate length (message $\alpha.3$). B creates an encrypted component of the appropriate form, and tries forwarding this to the server in message $\alpha.4$, but the intruder intercepts this message. Then the intruder starts a second run, β, again impersonating A, so as to use B as an oracle. He sends, in message $\beta.1$, the nonce N_B that B chose for run α. B returns a nonce (message $\beta.2$) and the intruder again responds with a bit-string (message $\beta.3$). When B tries to send message $\beta.4$ to the server, the intruder intercepts it again. This message is precisely of the form expected by B in message $\alpha.5$, hence the intruder forwards this message to B, and B accepts N_B' as a session key. Finally, B sends an appropriate message $\alpha.6$ and the intruder returns the component $[N_B]_{N_B'}^{\leftrightarrow}$ (message $\alpha.7$), which he can produce because he knows the "key" N_B', thus completing the handshake.

Finding the attack There are three protocol roles in the Woo and Lam protocol, A, B and the server S. The attack we are looking for spans two parallel sessions. In practice, we need a semibundle with at least two responder roles and no initiator roles, and it turns out that we don't have to specify any initiator nor server roles (adding one or more initiator or server roles to the semibundle would not prevent the system from finding the attack, however, it would slow it down; the same applies to the adding of more responder roles). The Prolog code is the following (we also specify the semibundle):

```
resp(A,B,N_A,N_N,K_AB,K_AS,K_BS,[
    recv([A,N_A]),
    send([B,N_B]),
    recv([A,[B,[N_A,N_B]]]+K_AS),
    send([([A,[B,[N_A,N_B]]]+K_AS),
          ([A,[B,[N_A,N_B]]]+K_BS)
         ]),
    recv([([B,[N_A,[N_B,K_AB]]]+K_AS),
          ([A,[N_A,[N_B,K_AB]]]+K_BS)
         ]),
    send([([B,[N_A,[N_B,K_AB]]]+K_AS),
          ([N_A,N_B]+K_AB)
         ]),
    recv(N_B+K_AB)
    ]).
semibundle([Sb1,Sb2]) :-
    resp(a,b,N_A1,n_B1,K_AB1,K_AS,k_BS,Sb1),
    resp(a,b,N_A2,n_B2,K_AB2,K_AS,k_BS,Sb2).
```

By running this code, we obtained Lowe's attack:

```
trace:
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), recv([a,b])]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), send([b,n_B1])]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), recv([a, [b,
```

```
[b, n_B1]]] + _1154)]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), send([[a, [b,
[b, n_B1]]] + _1154, [a, [b, [b, n_B1]]] + k_BS])]
[responder(a, b, n_B1, n_B2, _1318, _1154, k_BS), recv([a,n_B1])]
[responder(a, b, n_B1, n_B2, _1318, _1154, k_BS), send([b,n_B2])]
[responder(a, b, n_B1, n_B2, _1318, _1154, k_BS), recv([a,
[b, [n_B1, n_B2]]] + _1154)]
[responder(a, b, n_B1, n_B2, _1318, _1154, k_BS), send([[a,
[b, [n_B1, n_B2]]] + _1154, [a, [b, [n_B1, n_B2]]] + k_BS])]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), recv([[b, [b,
[n_B1, n_B2]]] + _1154, [a, [b, [n_B1, n_B2]]] + k_BS])]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), send([[b, [b,
[n_B1, n_B2]]] + _1154, [b, n_B1] + n_B2])]
[responder(a, b, b, n_B1, n_B2, _1154, k_BS), recv(n_B1+n_B2)]
reached state:
[[responder(a, b, b, n_B1, n_B2, K_AS, k_BS), []],
[responder(a, b, n_B1, n_B2, K_AB2, K_AS, k_BS), [recv([[b,
[n_B1, [n_B2, K_AB2]]] + K_AS, [a, [n_B1, [n_B2, K_AB2]]] +
k_BS]), send([[b, [n_B1, [n_B2, K_AB2]]] + K_AS, [n_B1,
n_B2] + K_AB2]), recv(n_B2 + K_AB2)]]]
```

Here, it is worth noticing that the system chooses the term `recv([a, [b, [b, n_B1]]] + _1154)` as a possible bit-string of proper length (introduced by Lowe). Here, _1154 is a variable, representing an infinite number of messages, all possible bit-strings. Also, notice that this is a partial run: in fact, the second strand is not completed. Inspecting the output, we can see that the system terminates right after message $\beta.4$.

The Original System To emphasize the practical importance of Proposition 2 we now show what needs to be done to detect the above attack in the original system MS. In that context, the only way of finding the flaw is by employing *exactly* two responder strands, in which the second strand contains only and exactly the first four events:

```
responder(A,B,N_A,N_B,K_AB,K_AS,K_BS,[
   recv([A,N_A]),
   send([B,N_B]),
   recv([A,[B,[N_A,N_B]]]+K_AS),
   send([([A,[B,[N_A,N_B]]]+K_AS),
        ([A,[B,[N_A,N_B]]]+K_BS)
        ]),
   recv([([B,[N_A,[N_B,K_AB]]]+K_AS),
        ([A,[N_A,[N_B,K_AB]]]+K_BS)
        ]),
   send([([B,[N_A,[N_B,K_AB]]]+K_AS),
        ([N_A,N_B]+K_AB)
        ]),
   recv(N_B+K_AB)
]).
```

```
responder_incomplete(A,B,N_A,N_B,K_AB,K_AS,K_BS,[
   recv([A,N_A]),
   send([B,N_B]),
   recv([A,[B,[N_A,N_B]]]+K_AS),
   send([([A,[B,[N_A,N_B]]]+K_AS),
         ([A,[B,[N_A,N_B]]]+K_BS)])
]).
```

This is clearly *ad-hoc*.

5 Benchmarks

Comparison with MS We use the Bilateral Key Exchange with Public Key
protocol (BKEPK) (described in the Clark and Jacob library [16]), and checked
secrecy of nonces. The protocol proceeds as follows:

Message 1. $B \rightarrow A : B, [N_B, B]^{\rightarrow}_{pk(A)}$
Message 2. $A \rightarrow B : [h(N_B), N_A, A, K_{AB}]^{\rightarrow}_{pk(B)}$
Message 3. $B \rightarrow A : [h(N_A)]^{\leftrightarrow}_{K_{ab}}$

Our test-bed was a Pentium IV 1.9GHz running Linux 2.4.7-10 and ECL^iPS^e
5.2. The comparison is shown in the following table. To make a fair comparison
and avoid listing each partial run as a possible solution, we forced the system
to find a secrecy flaw, which, since we are dealing with a non-flawed protocol,
causes all possible partial runs to be explored.

# strands	description	Our system (time (s))	MS (time (s))
6	2A2B2T	0.38	3.20
7	3A2B2T	2.61	42.52
8	3A2B3T	2.84	324.22
7	3A3B1T	236.23	991.20
8	3A3B2T	277.23	6865.70
9	3A3B3T	316.20	57635.48

The description field gives an idea of the semibundle tested, e.g. 2A2B2T
means that we are including 2 A roles, 2 B roles and checking secrecy of two
nonces (one for each session).

Operationally speaking, the main difference between our system and MS lies
in the fact that we check the constraints as soon as possible while producing the
run. This is in line with our earlier paper [17] (although the system presented here
has better performance). In MS first all (complete) runs are generated and then
it is checked which runs yield a solvable constraint: in our system we anticipate
failure, therefore reducing the search space. A more detailed comparison of the
efficiency of both systems is future work.

Experiments with protocols from Clark and Jacob library [16] We
ran our system succesfully on several flawed protocols from the Clark and Ja-

cob library [16]. The following table lists the performance and the attacks found.

Protocol	Attack	time (s)
Encrypted Key Exchange	Parallel Session	0.07
ISO symmetric key 1-pass unilateral auth	Replay	0.04
ISO public key 2-pass mutual authentication	Replay	0.11
Needham-Schroeder Symmetric Key	Man-in-the-middle	0.09
Needham-Schroeder Public Key	Man-in-the-middle	0.06
Needham-Schroeder-Lowe Public Key	Type Flaw	0.07
Neuman Stubblebine	Type Flaw	0.27
Otway-Rees	Type Flaw	0.22
SPLICE	Replay	0.21
Woo-Lam Mutual Authentication	Parallel Session	30.96
Yahalom with Lowe's alteration	Type Flaw	0.08

6 Conclusions

We have developed a constraint-based system for protocol verification based on the work of Millen and Shmatikov (MS) [30]. Our contribution is threefold:

- We have defined a new semantics which is more intuitive and yields to a considerably more efficient implementation, which can make the difference between days and minutes. This means that our system allows in practice the verification of more complex protocols and sessions than MS.
- We have a *monotonic system*: any (authentication or secrecy) attack that can be found using a semibundle S can be found also when using a semibundle S', provided that $S \preceq S'$.
  This has direct impact on the practicality of the system. For instance, our system allows for the automatic detection of attacks that rely on partial strands. While there are many such attacks (all attacks presented in this paper, for instance, rely on partial runs), the only way to detect them with MS is by providing the system with an ad-hoc semibundle that precisely reflects the partial run that exhibits the flaw.
  Suppose one wants to check seriously a protocol involving agents A and B for secrecy flaws, in presence of – say – maximal two parallel sessions. While with our system, thanks to Proposition 2, it is sufficient to run the tool on a semibundle S containing four honest strands (plus the secrecy strands), with MS one would have to try all semibundles $S' \preceq S$, which is infeasible.
- We have extended the system with the possibility of applying explicit checks. As shown in Section 3 this allows to model more complex protocols. In particular, we needed this feature when we used the system to analyze some non-repudiation protocols, among which Zhou-Gollmann's [41].

The system we presented here seems to be very fast. It is considerably faster than MS, which is well-known for its speed. In the near future, we intend to test its performances against other "fast" systems such as that of Chevalier and Vigneron [13] and of Basin [5].

7 Related Work

In previous work [17] we have defined a tool for protocol verification based on a *proof system*; the system presented here is radically different in syntax, semantics and implementation, and in particular it is much faster.

Several *logic*-based approaches have been applied to verification problems of security protocols. Among them, trace semantics is at the basis of the verification methods proposed, e.g., in [33, 34, 6, 5]. In [34], Paulson models a protocol in presence of an intruder as an inductively defined set of traces, and uses Isabelle and HOL to prove *interactively* the absence of attacks. Paulson's approach works on an infinite-state search space. Our approach is actually closer to Basin's method [5], where *lazy data structures* are used to generate the infinite-search tree representing protocol traces in a *demand-driven* manner. To limit state explosion, however, Basin applies heuristics that prune the generated tree. Here – on the contrary – no heuristics is needed. Trace semantics were also used for model-checking based analysis. In this context, we find the work of Lowe [25, 27], who first identified and fixed the flaws in the Needham-Schroeder protocol, and the work of Mitchell et al. [32]. As pointed out e.g., by Jacquemard, Rusinowitch and Vigneron [23], these approaches require ad-hoc solutions for limiting the search space. In contrast, our approach terminates for all protocols, provided that we limit the number of parallel sessions. Fiore and Abadi [20] and Boreale [8] also present algorithms for computing symbolic traces of infinite-state protocols. Automatic methods for rapid prototyping and analysis of protocols have also been presented. We mention Denker, Meseguer and Talcott [18] using Maude, and Mitchell, Mitchell and Stern [32] using Murϕ.

In a different approach, Cervesato et al. [11] use a formalism based on multiset-rewriting (linear logic) to specify protocol rules and actions of intruders. A multiset rewriting formalism has also been used by Rusinowitch et al. by implementing it in OCAML in the CASRUL system [13], and, in the later [23] by processing the rewrite rules in the theorem-prover daTac. Conceptually, our system shares some features with that developed (independently) by Chevalier and Vigneron [15, 14]. As in Chevalier and Vigneron [15, 14], we deal with communication in a constraint-based way, in which the intruder only checks if he is able to generate a certain message. On the other hand, Chevalier and Vigneron [15, 14] rely on a multiset rewriting formalism.

In addition to MS a few works have appeared that employ constraint-solving techniques. Typically, the task of the intruder is brought down to that of checking whether certain messages are *contrivable*. Among these works we find Rusinowitch and Turuani [37] and Huima's [22]. In the former the authors also demonstrate that the problem of finding an attack *in the case of bounded parallel sessions* is NP-complete. In our work, we require the intruder only to *check* whether he is able to generate a certain message, while these approaches do this in a more radical way: the checking phase is postponed *after* that the whole trace has been completed (we check it after the next protocol step has been carried out).

Logic programming languages have been applied to specify security protocols in several different ways. Meadow's NRL Protocol Analyzer [29] performs

a reachability analysis using state-enumeration enriched by lemmata proved by induction. This way, NRL can cope with a potential infinite search space. Our approach differs from this previous work in the first place in that NRL explores the search space in a backward fashion, i.e., starting from the unwanted situation and working backwards to find possible traces leading to it.

A logic programming language is used by Aiello and Massacci [3] but using a different perspective, i.e., with *stable semantics*, to specify and debug protocols. In this setting knowledge, protocol rules, intruder capabilities and objectives are specified in a declarative way. Finally, Blanchet [7] uses Prolog to specify *conservative abstractions* that can be used to prove security protocols free from attacks. Intuitively, Horn clauses are used here as *constructors* and *destructors* for the messages exchanged on the net and intercepted by the intruder. Abadi and Blanchet [2] relate this approach to that of using *types* for guaranteeing the secrecy of communication [1], an approach substantially different from ours. The verification procedure combines aspects of *unfolding* and *bottom-up* evaluation (following, however, a depth-first strategy). A way of better understanding the relationships with our approach would be to extend the bottom-up evaluation strategy to the multi-conclusion setting as suggested by Bozzano [9].

Most of the above work can analyze a finite number of protocol sessions. Noteworthy exception to this rule are: firstly, the approaches based on Theorem Proving (Paulson [33, 34]), together with the NRL analyzer, that require human intervention to limit the search space. Secondly, the works based on some form of abstraction, which does not guarantee completeness, like Abadi and Blanchet [2] (which is based on types) and Schneider's work on *rank functions* [38]. Blanchet also developed a protocol verifier [7], which guarantees termination but has the disadvantage that false attacks may be reported. Finally, there is work in which the analysis might not terminate, like Basin [5] (however the heuristics they apply are successful in most cases).

References

1. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In F. Honsel and M. Miculan, editors, *Proc. Foundation of Software Science and Computation Structures (FoSSaCS 2001)*, volume 2030 of *LNCS*, pages 25–41. Springer-Verlag, 2001.
2. M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *29th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 33–44, Portland, Oregon, 2002. ACM Press.
3. L. C. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *Transactions on Computational Logic*, 2(4):542–580, 2001.
4. K. R. Apt. *From Logic Programming to Prolog*. International Series in Computer Science. Prentice Hall, 1997.

5. D. Basin. Lazy infinite-state analysis of security protocols. In R. Baumgart, editor, *Secure Networking - CQRE (Secure) '99, International Exhibition and Congress*, volume 1740 of *LNCS*, pages 30–42. Springer-Verlag, 1999.

6. G. Bella and L. C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In J.-J. Quisquater, editor, *Proc. 5th European Symposium on Research in Computer Security*, volume 1485 of *LNCS*, pages 361–375, Louvain-la-Neuve, Belgium, 1998. Springer-Verlag.

7. B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In S. Schneider, editor, *Proc. 14th IEEE Computer Security Foundations Workshop*, 2001.

8. M. Boreale. Symbolic trace analysis of cryptographic protocols. In *28th Colloquium on Automata, Languages and Programming (ICALP)*, LNCS, pages 667–681. Springer-Verlag, 2001.

9. M. Bozzano. Ensuring security through model checking in a logical environment (preliminary results). In G. Delzanno, S. Etalle, and M. Gabbrielli, editors, *Proc. Workshop on Specification, Analysis and Validation for Emerging Technologies (SAVE01)*, 2001.

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

11. I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *PCSFW: Proc. 12th Computer Security Foundations Workshop*, pages 55–69. IEEE Computer Society Press, 1999.

12. I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *PCSFW: Proc. 13th Computer Security Foundations Workshop*, pages 35–51. IEEE Computer Society Press, 2000.

13. Y. Chevalier, F. Jacquemard, M. Rusinowitch, M. Turuani, and L. Vigneron. CAS-RUL web site. `http://www.loria.fr/equipes/protheo/SOFTWARES/CASRUL/`.

14. Y. Chevalier and L. Vigneron. A tool for lazy verification of security protocols. In *Proc. 16th IEEE International Conference Automated Software Engineering*, 2001.

15. Y. Chevalier and L. Vigneron. Towards efficient automated verification of security protocols. In *Proc. VERIF01, Verification Workshop in conjunction with IJCAR*, pages 19–33, 2001.

16. J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. `http://www.cs.york.ac.uk/ jac/papers/drareview.ps.gz`, 1997.

17. G. Delzanno and S. Etalle. Proof theory, transformations, and logic programming for debugging security protocols. In A. Pettorossi, editor, *Proc. Eleventh International Workshop on Logic Program Synthesis and Transformation – LOPSTR 2001*, LNCS, pages 76–91. Springer-Verlag, 2002.

18. G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *In Proc. of Workshop on Formal Methods and Security Protocols*, 1998.

19. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

20. M. Fiore and M. Abadi. Computing symbolic models for verifying cryptographic protocols, 2001.

21. J.C. Herzog F.T. Fabrega and J.D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings of The 1998 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1998.

22. A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. Workshop Formal Methods and Security Protocols (FLOC 1999)*, 1999.

23. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In M. Parigot and A. Vonkorov, editors, *Proc. LPAR: International Conference on Logic for Programming and Automated Reasoning*, number 1995 in LNCS, pages 131–160. Springer-Verlag, 2000.

24. J. W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, 1987. Second edition.

25. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, Berlin Germany, 1996.

26. G. Lowe. Some new attacks upon security protocols. In *PCSFW: Proceedings of The 9th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1996.

27. G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW '97)*, pages 18–30. IEEE, 1997.

28. C. Meadows. Formal verification of cryptographic protocols: A survey. In J. Pieprzyk and R. Safavi-Naini, editors, *Advances in Cryptology – ASIACRYPT '94*, LNCS, pages 133–150. Springer-Verlag, 1995.

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

30. J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. 2001 ACM Conference on Computer and Communication Security*, pages 166 – 175. ACM press, 2001.

31. J. K. Millen, S. C. Clark, and S. B. Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, 13(2):274–288, February 1987. Special Issue on Computer Security and Privacy.

32. J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proceedings of the 1997 Conference on Security and Privacy*, pages 141–153. IEEE Press, 1997.

33. L. C. Paulson. Mechanized proofs of security protocols: Needham-Schroeder with public keys. Technical Report 413, University of Cambridge, Computer Laboratory, January 1997.

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

35. A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *IEEE Symposium on Foundations of Secure Systems*, 1995.

36. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1999.

37. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is np-complete. In S. Schneider, editor, *Proc. 14th IEEE Computer Security Foundations Workshop*, 2001.

38. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. Modelling and analysis of security protocols, 2001.

39. D. X. 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.

40. T.Y.C. Woo and S. S. Lam. A lesson on authenticated protocol design. *Operating Systems Review*, 28(3):24–37, 1994.

41. J. Zhou and D. Gollmann. An efficient non-repudiation protocol. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97)*, pages 126–132, 1997.