Paper

Process calculi and the verification of security protocols

Michele Boreale and Daniele Gorla

Abstract — Recently there has been much interest towards using formal methods in the analysis of security protocols. Some recent approaches take advantage of concepts and techniques from the field of process calculi. Process calculi can be given a formal yet simple semantics, which permits rigorous definitions of such concepts as "attacker", "secrecy" and "authentication". This feature has led to the development of solid reasoning methods and verification techniques, a few of which we outline in this paper.

Keywords — cryptographic protocols, Dolev-Yao model, observational equivalence, process calculi, spi calculus.

1. Introduction

Security protocols have become an essential ingredient of communication infrastructures. When executed in a hostile environment, these protocols may be subject to a number of attacks, that can compromise the security of the data being exchanged over a network. An attacker might typically learn a piece of information which is supposed to remain secret, or it might fool an agent into accepting a compromised key as authentic. Proving a protocol resistant to such attacks is notoriously a difficult task. In the last decade, formal methods have been successfully used to analyse security protocols, sometimes uncovering flaws in protocols that were thought to be correct.

The BAN logic [12] was one of the first, partially successful attempts at using formal methods in the field of security. Later on, finite-state model checking has been extensively used (see e.g. [21, 26]). Some recent developments of formal methods stem from concepts well established in the field of process calculi. In particular, Abadi and Gordon have proposed the *spi-calculus* [3] by elaborating on Milner, Parrow and Walker's π-calculus [24], a process language based on synchronous message passing. The spi-calculus extends the π-calculus with cryptographic primitives, thus allowing the description of security protocols as systems of concurrent processes that can exchange encrypted data. The main advantage of this approach is that process calculi can be given formal yet simple semantics that permit rigorous definitions of such notions as "attacker", "secrecy" and "authentication". Another distinguishing feature of the π-calculus is its reliance on the powerful scoping constructs of the π-calculus to get a clean formalization, at a linguistic level, of such concepts as "nonce", and "newly generated key". In a sense, the spi-calculus improves both the BAN logic, which provides formal reasoning rules but not an operational model, and finite-state methods, which provide a precise operational model but not a convenient basis for formal reasoning. These features have led to the development of solid reasoning techniques and verification methods (e.g. [4, 5, 7, 8, 10]), a few of which we will survey in this paper.

In Section 2 we give a brief overview of the spi-calculus, mainly concentrating on syntax and informal explanation of its operators. Section 3 is devoted to presenting a simplified version of the *Kerberos* protocol [20], which will serve as a running example. While this small protocol is well suited for illustrating the key ideas of the approaches presented here, the reader should be warned that proofs for more sophisticated, in particular multi-session, protocols require a higher degree of ingenuity (see [3, 10]). In Sections 4 and 5 two formal semantics of the spi-calculus are outlined: the first is based on *observational equivalences*, the second is centered around the idea of *trace analysis*. Based on these semantics, rigorous reasoning principles and verification methods are described. Section 6 compares the presented approaches, while Section 7 contains a few concluding remarks and comparison with related work.

2. An outline of the spi-calculus

In this section, we intend to give an informal account of the spi-calculus, by concentrating on syntax and intuitive explanation. The reader is referred to [3, 10] for full technical details.

There are several versions of the spi-calculus. In the rest of this paper, we will consider a variant supporting shared-key cryptography only. This limited language is sufficient to illustrate the key ideas of the approach, while avoiding many technicalities.

Syntax. The syntax of the language is summarized in Table 1. A countable set $\mathcal{N}$ of *names* $a, b \ldots, h, k, \ldots, x, y, z \ldots$ is assumed. Names can be used as variables, communication channels, primitive data or keys: we do not distinguish between these four kinds of objects (notationally, we prefer letters $h, k, \ldots$ when we want to stress the use of a name as a key). Messages are built via pairing and shared-key encryption. In particular, $\{M\}_k$ represents the ciphertext obtained by encrypting M under key k, using a shared-key encryption system. An informal explanation of the process operators might be the following:

- **0** is the process that does nothing;
- $\tau.P$ does one internal computation step (we do not care precisely what), and then proceeds like P;

- $a(x).P$ waits for a message on channel a and then binds it to variable x within P;
- $\overline{a}\langle M\rangle.P$ sends message M on channel a and then behaves like P;
- $[M=N]P$ behaves like P if the M equals N, otherwise it is stuck;
- $\texttt{case}\ M\ \texttt{of}\ \{y\}_k\ \texttt{in}\ P$ attempts decryption of M using k as a key: if the decryption succeeds, i.e. if $M=\{M'\}_k$ for some M', then M' is bound to variable y within P, otherwise the whole process is stuck;
- $\texttt{pair}\ M\ \texttt{of}\ \langle x,y\rangle\ \texttt{in}\ P$ attempts splitting M; if this is possible, i.e. if M is a pair $\langle M',N'\rangle$, the two components M' and N' are bound, respectively, to variables x and y within P, otherwise the whole process is stuck;
- $(\nu b)P$ creates a new name b which is only known to P;
- $P+Q$ can behave either as P or Q: the choice may be triggered either by the environment or by internal computations of P or Q;
- $P|Q$ is the parallel execution of P and Q;
- $!P$ can be thought of as unboundedly many copies of P running in parallel, i.e. as $P|P|P|\cdots$.

Table 1
Syntax of the calculus

$a,b\ldots,h,k,\ldots,x,y,z\ldots$	*names* $\mathscr{N}$
$M,N ::= a \mid \langle M,N\rangle \mid \{M\}_k$	*messages* $\mathscr{M}$
$P,Q ::=$	*processes* $\mathscr{P}$
$\mathbf{0}$	(*null*)
$\mid \tau.P$	(*internal action*)
$\mid a(x).P$	(*input prefix*)
$\mid \overline{a}\langle M\rangle.P$	(*output prefix*)
$\mid [M=N]P$	(*match*)
$\mid \texttt{case}\ M\ \texttt{of}\ \{y\}_k\ \texttt{in}\ P$	(*decryption*)
$\mid \texttt{pair}\ M\ \texttt{of}\ \langle x,y\rangle\ \texttt{in}\ P$	(*splitting*)
$\mid (\nu b)P$	(*restriction*)
$\mid P+Q$	(*choice*)
$\mid P\mid Q$	(*parallel*)
$\mid !P$	(*replication*)

For the sake of simplicity, we are not considering integer data values present in [3], nor the general form of boolean guard used in [10]. In the definition of this language there are a few implicit assumptions on the underlying shared-key encryption system. We try to make them explicit below:

1) a plaintext M encrypted under a key k can only be decrypted using k; if the attacker does not know k, he/she cannot guess or forge this key (*perfect encryption*);
2) the only way to produce a ciphertext that looks like $\{M\}_k$ is to encrypt M under k;
3) there is enough redundancy in the structure of messages to tell whether a given ciphertext is correctly decrypted with a given key.

The first assumption implies that we can say nothing about attacks that exploit probabilistic or statistical analysis, which may arise in practice, as showed in [28]. In fact, we are concentrating on high-level, logical properties of protocols. The second assumption is an abstraction of the small probability, for real cryptosystems, that different ⟨plaintext, key⟩ pairs collide onto the same ciphertext. The third assumption is in practice implemented by attaching a cryptographic checksum to every plaintext before encryption.
We fix now a few notational shorthands that will be used in the remainder of the paper:

- $a(x).\cdots$ is a binder for x, $\texttt{case}\ \cdot\ \texttt{of}\ \{y\}_k\ \texttt{in}\cdots$ is a binder for y, $\texttt{pair}\ \cdot\ \texttt{of}\ \langle x,y\rangle\ \texttt{in}\ \cdots$ is a binder for x and y and restriction $(\nu b)\cdots$ is a binder for b. We shall also say that x, y and b are *bound* names. Bound names can be renamed to fresh names without affecting the meaning of a process term. We shall always assume that bound names are distinct from each other and from the names that are not bound.
- Names that are not bound are *free*. We use the notation $P(x)$ to emphasize that name x may occur free (i.e. not in the scope of any binder for x) in P and, for any message M, write $P(M)$ to abbreviate $P[^M/_x]$ i.e. P with each free occurrence of x replaced by of M. The set of free names of a process P will be written as $fn(P)$.
- $[M=N,M'=N']$ stands for two consecutive matchings $[M=N][M'=N']$. Similarly, we shall use the shorthands $(\nu a,b)P$ for $(\nu a)(\nu b)P$ and $\texttt{pair}\ M\ \texttt{of}\ \langle x,y,z\rangle\ \texttt{in}\ P$ for $\texttt{pair}\ M\ \texttt{of}\ \langle x,l\rangle\ \texttt{in}\ \texttt{pair}\ l\ \texttt{of}\ \langle y,z\rangle\ \texttt{in}\ P$. The tilde symbol $\tilde{}$ will be used to denote vectors of objects.

A small example illustrates the use of the calculus for describing cryptographic protocols.

Example. Consider the simple protocol where two principals A and B share a private key k. A wants to send B a datum d encrypted under k, through a public channel c. B accepts any message encrypted with k that is sent along c:

$$A \to B : \{d\}_k \text{ on channel } c.$$

This informal notation can be translated into the spi-calculus process P defined as follows:

$$
\begin{array}{lcl}
A & \stackrel{\text{def}}{=} & \overline{c}\langle \{d\}_k \rangle .\mathbf{0} \\
B & \stackrel{\text{def}}{=} & c(x).\,\mathtt{case}\ x\ \mathtt{of}\ \{y\}_k\ \mathtt{in}\ F(y) \\
P & \stackrel{\text{def}}{=} & (\nu k)(A|B).
\end{array}
$$

A stops after outputing $\{d\}_k$ on c. B picks up any message from c and then tries to decrypt it using k. If decryption succeeds, the result is bound to variable y within $F(y)$. The latter is some expression describing the subsequent behaviour of B, depending on the result of the decryption, y. The whole protocol P is the parallel composition $A|B$, with the restriction (νk) indicating that the key k is only known to A and B.

On restricted names. The restriction operator plays a crucial role in the spi-calculus. $(\nu k)P$ makes the name k *private* to P. This resembles declarations of local variables in structured programming languages. There is one crucial difference, however: in spi-calculus, a restricted name can be *exported* outside its original scope, while remaining distinct from every name of the recipient. As such, the restriction operator is ideal for modelling those "fresh unguessable quantities" (like random numbers) that are an important ingredient of many cryptographic protocols. The following equation, for instance, explains the creation of a nonce n and its transmission from one principal to another, along a private channel c:

$$
(\nu c)\Big(\big((\nu n)\,\overline{c}\,\langle n\rangle .A\big) \mid c(x).B(x)\Big) = \tau .(\nu c, n)\big(A \,|\, B(n)\big)\,.
$$

The symbol $=$ above can be given a precise meaning in terms of observational semantics, as we shall see in Section 4. Informally, this equation says that the consumption of complementary input and output prefixes $\big(c(x).$ and $\overline{c}\langle n\rangle .\big)$ gives rise to an internal communication (represented by the $\tau.$ prefix) in which n is communicated. This also causes the scope of the restriction (νn) to be extended so as to include B. The scope extension is capture-avoiding, in the sense that n is automatically renamed if it happens to clash with some name in B. This phenomenon is called *scope extrusion* of name n.

A slightly more complicated equation holds when c is a public, rather than private, channel. In this case, the equation also explains the possible interaction of the two principals with the external environment along c.

3. The BAN Kerberos protocol

We shall illustrate the techniques presented in later sections on the version of the Kerberos protocol considered by Burrows, Abadi and Needham in [12]. This section is devoted to an informal presentation of this protocol.

Consider a system where two agents A (the initiator) and B (the responder) share two long-term secret keys, k_{AS} and k_{BS} respectively, with a server S. The protocol is designed to set up a new secret session key k_{AB} between A and B. Informally, the protocol can be described as follows:

$$
\begin{array}{lcl}
A \longrightarrow S & : & A, B \\
S \longrightarrow A & : & \big\{T, k_{AB}, B, \{T, k_{AB}, A\}_{k_{BS}}\big\}_{k_{AS}} \\
A \longrightarrow B & : & \{T, k_{AB}, A\}_{k_{BS}}, \{A, n_A\}_{k_{AB}} \\
B \longrightarrow A & : & \{n_A\}_{k_{AB}}.
\end{array}
$$

In the first message, A starts the protocol by simply communicating to S his intention to establish a new connection with B. In the second message, S generates a fresh key k_{AB} and inserts it into an appropriate certificate, which is sent to A. The certificate uses a timestamp T, meant to assure A and B about the freshness of the message: this is to counter attacks based on replays of old messages. In the third message, A extracts B's part of the certificate $\big(\{\cdots\}_{k_{BS}}\big)$ and forwards it to B, together with some challenge information containing a new nonce n_A. The fourth message is B's response to A's challenge: the presence of n_A is meant to assure A he is really talking to B.

In the next two sections, relying on two different techniques, we shall verify one session configuration of this protocol, under the hypothesis that an old session key $k_{\mathtt{old}}$ between A and B has been compromised. We shall not consider the multi-session case, which requires a more complex analysis. For the sake of simplicity, we shall also suppose that the protocol is always initiated by A and that the responder is always B.

4. Observational equivalences

Following [3], a powerful way of expressing authentication properties of a security protocol P is to require that P is *equivalent* to a process Q that, by definition, exhibits the desired behaviour (e.g., Q never accepts non-authentic messages). Secrecy as well can be expressed via this notion of equivalence. For example, let $P(d)$ be a process in which a secret datum d is exchanged, properly encrypted, along a public channel. A way of asserting that $P(d)$ keeps d secret is requiring that $P(d)$ be equivalent to $P(d')$, for every other d'. An appropriate notion of equivalence is here *may-testing* [3, 9, 14]. Its intuition is precisely that *no* external observer (which in the present setting can be read as "attacker") can notice any difference when, e.g., running in parallel with $P(d')$ or $P(d)$. Formally, we define an observer as a process that is possibly capable of a distinct "success" action ω; the latter is used to signal that the observed process has passed observer's test. If one interprets "passing a test" as "revealing a piece of information", then processes that may pass the same tests may potentially reveal the same information to external observers: as such, they should be considered equivalent from a security point of view. This also accounts for implicit information flow, by which an observer might extract useful information from the overall behaviour of a system.

In the definition below, $R \stackrel{\omega}{\Longrightarrow}$ means that R can execute zero or more internal computation steps, followed by an ω–action.

Definition 1 (may-testing). Two spi-calculus processes P and Q are *may-testing equivalent*, written $P \simeq Q$, if for every observer O, $P \mid O \stackrel{\omega}{\Longrightarrow}$ iff $Q \mid O \stackrel{\omega}{\Longrightarrow}$.

A similar intuition is supported by other contextual equivalences, like *barbed equivalence* [25]. While rigorous and intuitive, the definitions of these equivalences suffer from universal quantification over contexts (attackers), that makes equivalence checking very hard. It is then important to devise proof techniques that avoid such quantification.

Results in this direction are well-known for traditional process calculi. For example, both in CCS [14] and in the π-calculus [9], may-testing is easily proven to coincide with *trace equivalence*, which requires that two equivalent processes generate the same sequences of *actions* (I/O events). Similarly, barbed equivalence is proved to coincide with *early bisimulation*. The latter requires that each action of one process be "simulated" by the other, and that the target processes be still bisimilar. In this section we outline a way of obtaining similar results in the case of the spi-calculus; full details can be found in [10]. We then discuss a few resulting reasoning rules and apply them to the Kerberos protocol.

4.1. A labelled transition system for the spi-calculus

In non cryptographic calculi (like the π-calculus) processes and observers share the same knowledge of names. This means, in essence, that the external environment may enable any action that a process is willing to take. This is not true anymore when moving to the spi-calculus. In fact, consider the process P that sends a fresh name b encrypted with a fresh key k and then executes P'. This is written $(\nu\, b,k)\overline{c}\langle\{b\}_k\rangle.P'$. When an observer receives $\{b\}_k$, it does not acquire automatically the knowledge of b, because k is still secret. Thus, if P' is willing to input something at b (say $P' \stackrel{\text{def}}{=} b(x).P''$), the environment cannot satisfy P''s expectations. For this reason, execution traces *à la* π-calculus fail to capture the interactive behaviour of processes.

This discrepancy leads us to make the concept of *environment* explicit, as a record of the knowledge of names and keys that an external observer has acquired about a certain process. More precisely, we model an environment as a mapping σ from a set of variables to a set of messages. Intuitively, an environment is a set of locations named by distinct variables, where an observer (usually an attacker) will store information known. We want now to describe how the environment is modified by the actions performed by the process and how actions that the process can perform are constrained by the environment. To this purpose, we introduce an environment-sensitive labelled transition system (written e.s.–lts in the sequel), whose states are *configurations* $\sigma \triangleright P$, where σ is the current environment and P is a process. Transitions between configurations represent interactions between σ and P, and take the form

$$\sigma \triangleright P \stackrel{\mu}{\underset{\delta}{\longmapsto}} \sigma' \triangleright P',$$

where μ is the action of process P and δ is the complementary *environmental action*. More precisely, μ can be of three forms: an internal action – τ – an input – aM – or an output – $(\nu\widetilde{b})\overline{a}\langle M\rangle$. The latter makes explicit the private names $\widetilde{b}$ that are being extruded. Accordingly, the environmental action δ is a "no-action", an output or an input. Therefore, three kinds of transitions may arise:

1. The process performs an output and the environment an input. As a consequence, the environment's knowledge gets updated. For instance:

$$\sigma \triangleright P \stackrel{(\nu\widetilde{b})\overline{a}\langle M\rangle}{\underset{z(x)}{\longmapsto}} \sigma[M/x] \triangleright P',$$

   where $\sigma[M/x]$ is the update of σ with the new entry $[M/x]$, for a fresh variable x. Here, $\widetilde{b}$ is the set of private names the process extrudes. For the transition to take place, channel a must belong to the knowledge of σ, which in this case amounts to saying that $\sigma(z) = a$.

2. The process performs an input and the environment an output. Notice that messages from the environments cannot be arbitrary, but must be built via encryption, decryption, pairing and projection, from the messages recorded in σ, plus some fresh names the environment can create. Thus, a transition might be:

$$\sigma \triangleright P \stackrel{aM}{\underset{(\nu\widetilde{b})\overline{z}\langle\zeta\rangle}{\longmapsto}} \sigma[\widetilde{b}/\widetilde{b}] \triangleright P'.$$

   Here, $\widetilde{b}$ is the set of new names the environment has just created and added to its knowledge, while ζ is an expression describing how M has been built out of σ and $\widetilde{b}$. This expression uses the variables in the domain of σ. For example, if $\sigma = [c/x_1, k/x_2, \ \ldots\ ]$ and $M = \{c\}_k$, then ζ might be $\{x_1\}_{x_2}$, indicating that message M results from encrypting the x_1–entry using the x_2–entry as a key. Again, a must belong to the knowledge of σ, thus $\sigma(z) = a$.

3. The process performs an internal move and the environment does nothing:

$$\sigma \triangleright P \stackrel{\tau}{\underset{-}{\longmapsto}} \sigma \triangleright P'.$$

Having introduced the e.s.-lts, we can define a new equivalence on top of it. The equivalence should only relate configurations that exhibit equivalent environments. Informally, two environments are equivalent if there is no way of telling them apart by performing elementary operations (like projection, decryption, comparison and so on) on their entries. For instance, $\sigma \stackrel{\text{def}}{=} [a/x, b/y, \{a\}_k/z]$ and $\sigma' \stackrel{\text{def}}{=} [a/x, b/y, \{b\}_k/z]$ are equivalent, while $\sigma[k/w]$ and $\sigma'[k/w]$ are not, because k enables decryption of the z-entry, and then comparing the obtained cleartext with the first two

entries yields different results. A formalization of these concepts can be found in [10]; for our purposes, this informal explanation suffices. The taken point of view is that two equivalent configurations should exhibit the *same environmental actions*, no matter what the process actions are. These consideration lead to the definition below. We write $\Longmapsto$ for the reflexive and transitive closure of $\stackrel{\tau}{\longmapsto}$ (i.e., a sequence of zero or more $\stackrel{\tau}{\longmapsto}$ transitions) and, inductively, $\stackrel{s}{\Longmapsto}$ for $\Longmapsto\stackrel{\mu}{\longmapsto}\stackrel{s'}{\Longmapsto}$ when $s = \mu \cdot s'$ and $u = \delta \cdot u'$. With this notation we have:

Definition 2 (e.s. trace equivalence). Let σ_1 and σ_2 be equivalent environments. Given two processes P and Q, we write $(\ \sigma_1\ ,\ \sigma_2\ ) \vdash P \simeq_{\mathtt{tr}} Q$ if whenever $\sigma_1 \triangleright P \stackrel{s}{\Longmapsto} \sigma_1' \triangleright P'$ then there are s', σ_2' and Q' such that $\sigma_2 \triangleright Q \stackrel{s'}{\Longmapsto} \sigma_2' \triangleright Q'$ and σ_1' is equivalent to σ_2', and symmetrically for $\sigma_2 \triangleright Q$.

This definition highlights a major difference between the π-calculus and the spi-calculus. In the π-calculus "exact" correspondence is required between actions of two equivalent processes P and Q, in the sense that if P is capable of an α-action, then Q must be capable of α too. On the contrary, the presence of cryptography in the spi-calculus allows for a "looser" correspondence. In fact, encrypting two different messages with a secret key makes the two messages indistinguishable for any external observer. Hence, for example, the processes $P \stackrel{\text{def}}{=} (\nu\, k)\overline{c}\langle\{a\}_k\rangle.\,\mathbf{0}$ and $Q \stackrel{\text{def}}{=} (\nu\, k)\overline{c}\langle\{b\}_k\rangle.\,\mathbf{0}$ are equivalent, even though they do not perform the same (process) actions.

Trace equivalence avoids quantification over contexts and only requires considering transitions of the e.s.-lts. Thus, when compared to the contextual definition of may testing, trace equivalence make reasoning on processes much easier. The following theorem ensures that $\simeq_{\mathtt{tr}}$ is a sound and complete characterization of may-testing equivalence $\simeq$. We denote by ε_V the environment that acts as the identity on the set of names V.

Theorem 1. Let P and Q be spi-processes, and let $V = fn(P,Q)$. It holds that $(\ \varepsilon_V\ ,\ \varepsilon_V\ ) \vdash P \simeq_{\mathtt{tr}} Q$ iff $P \simeq Q$.

A similar result holds for barbed equivalence and an environment-sensitive version of bisimulation.

4.2. Sound reasoning principles

Trace equivalence can be used to justify some rules for syntax-driven reasoning, which are at the core of a sound and complete proof system for the spi-calculus [11]. The rules we are going to list are valid for both bisimulation and trace equivalence. Thus, in what follows, we shall generically write $(\ \sigma_1\ ,\ \sigma_2\ ) \vdash P = Q$ to mean that the configurations $\sigma_1 \triangleright P$ and $\sigma_2 \triangleright Q$ are equivalent, without specifying the actual equivalence.

Structural laws. Table 2 lists a few fundamental equations, mostly inherited from the π-calculus [23], that are valid for any "reasonable" process equivalence. Most of them have to do with "static" structure of processes. Usually, the last three equations are not included in structural equivalence; we have included them here because they are natural in a cryptographic setting. The least equivalence relation over process terms that contains these equations is denoted by $\equiv$ and called *structural equivalence*. One can easily prove the following rule sound:

Table 2
Structural equivalence

$$P + \mathbf{0} \equiv P \qquad P + Q \equiv Q + P$$
$$P + (Q + R) \equiv (P + Q) + R$$

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P$$
$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

$$P \mid !P \equiv !P$$

$$(\nu\, b)\mathbf{0} \equiv \mathbf{0}$$
$$(\nu\, a)(\nu\, b)P \equiv (\nu\, b)(\nu\, a)P$$
$$((\nu\, a)P) \mid Q \equiv (\nu\, a)(P \mid Q) \quad \text{if } a \notin fn(Q)$$

$$[M = M]P \equiv P \qquad (\nu\, n)[n = M]P \equiv \mathbf{0} \quad \text{if } M \neq n$$

$$\mathsf{case}\ \{N\}_k\ \mathsf{of}\ \{y\}_k\ \mathsf{in}\ P \equiv P[^N/_y]$$

$$\mathsf{pair}\ \langle M_1, M_2\rangle\ \mathsf{of}\ \langle x,y\rangle\ \mathsf{in}\ P \equiv P[^{M_1}/_x, ^{M_2}/_y]$$

$$\frac{P \equiv Q}{(\ \sigma\ ,\ \sigma\ ) \vdash P = Q}.$$

In our example of Section 4.3 we shall make extensive use of two laws derived from structural equivalence. The first one is the so called *extrusion law*:

$$\text{(EXTR)} \qquad \frac{k \notin fn(Q)}{(\ \sigma\ ,\ \sigma\ ) \vdash ((\nu\, k)P) \mid Q = (\nu\, k)(P \mid Q)}.$$

It states that, if a restricted name k of P does not occur in a process Q running in parallel with P, then the scope of the restriction can be extended so as to include Q.

The second law we shall use is actually a pair of laws (that we shall globally refer to as (MATCH)) can be derived from the structural laws for the matching predicate $[M = N]$. In what follows, we call *context* a process $C[\ \cdot\ ,\ldots,\ \cdot\ ]$ with n "holes" that can be filled with n terms, thus yielding a proper process:

(MATCH)

$$\frac{M \text{ is not a name bound by } (\nu n)\, C[\,\cdot\,]}{(\,\sigma\ ,\ \sigma\,) \vdash (\nu n)\, C[\,P + [n = M]Q\,] = (\nu n)\, C[\,P\,]}\,.$$

$$(\,\sigma\ ,\ \sigma\,) \vdash C[\,P + [M = M]Q\,] = C[\,P + Q\,]$$

Transitivity. We shall also widely use the obvious transitivity rule:

(TRANS)

$$\frac{(\,\sigma_1\ ,\ \sigma_2\,) \vdash P = Q \quad \wedge \quad (\,\sigma_2\ ,\ \sigma_3\,) \vdash Q = R}{(\,\sigma_1\ ,\ \sigma_3\,) \vdash P = R}\,.$$

Parallel composition. The spi-representation of a security protocol is usually built up by putting in parallel a few simple spi-processes, corresponding to the principals involved in the protocol. A desirable property of each process calculus is that equivalence proofs can be done *compositionally*, i.e. by proving equivalences between subprocesses and then combining together such partial results to get the wanted claim. Unluckily, observational equivalences on the of spi-calculus are not closed under some operators, notably parallel composition. In particular, a naive law like

$$\frac{(\,\sigma_1\ ,\ \sigma_2\,) \vdash P = Q \quad \wedge \quad (\,\sigma_1\ ,\ \sigma_2\,) \vdash R = S}{(\,\sigma_1\ ,\ \sigma_2\,) \vdash P \,|\, R = Q \,|\, S}$$

is not valid. This is due to the interplay between cryptography and private names. As we have already shown at the beginning of Subsection 4.1, a private name k can be extruded and hence become free, without this implying that k is learnt by any observer. As a consequence, we are sometimes confronted with equivalences like: $(\,\sigma_1\ ,\ \sigma_2\,) \vdash \overline{c}\langle\{a\}_k\rangle.P_1 = \overline{c}\langle\{b\}_k\rangle.P_2$ where both σ_1 and σ_2 know a, b and c, but neither knows k. In general, this kind of equations are not preserved by parallel composition. For instance, when putting $R \stackrel{\text{def}}{=} \overline{c}\langle k\rangle.\mathbf{0}$ in parallel to both sides of the previous relation, the equivalence breaks down. The reason is that R may provide an observer with the key k to open $\{a\}_k$ and $\{b\}_k$, thus enabling a distinction between these two messages. Similar problems arise from the output prefix (see [11] for a general discussion about problems arising with compositional techniques in the spi-calculus). Fortunately, a more restrictive formulation does hold. Let us denote by $R\sigma$ the result of replacing each name x occurring free in R by $\sigma(x)$. Then we have:

(PAR)
$$\frac{(\,\sigma_1\ ,\ \sigma_2\,) \vdash P = Q}{(\,\sigma_1\ ,\ \sigma_2\,) \vdash P \,|\, R\sigma_1 = Q \,|\, R\sigma_2}$$

$$\text{if } fn(R) \subseteq \mathrm{dom}(\sigma_1) = \mathrm{dom}(\sigma_2)\,.$$

The side condition reduces the set of processes that can be composed with P and Q, by requiring that the composed processes are consistent with the knowledge available to σ_1 and σ_2. In spite of this limitation, the rule allows for non trivial forms of compositional reasoning, as shown in [11].

case elimination. A common situation for an agent involved in a protocol is waiting for a message and then trying to decrypt it using a key k. This is written as $P \stackrel{\text{def}}{=} p(x).\,\mathsf{case}\,x\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\,P'$. Now, suppose that, in some configuration, P comes equipped with an environment $\sigma \stackrel{\text{def}}{=} \sigma'[\{b\}_k/w]$, such that neither k nor $\{\cdot\}_k$ appears in σ'. Before P evolves, the only message of the form $\{\cdot\}_k$ that σ can produce is $\{b\}_k$. In other words the only message P can receive and then properly decrypt using k is $\{b\}_k$. Thus the behaviour of P in σ is equivalent to $p(x).[x = \{b\}_k]Q[b/y]$. The rule below generalizes this reasoning. We use the notation $\sum_{i=1}^{n} P_i$ to denote the process $P_1 + \ldots + P_n$ (this notation is well-defined since the non-deterministic choice is associative).

(CASE)

$$\begin{aligned}(\,\sigma\ ,\ \sigma\,) \vdash\ & (\nu \widetilde{h},k)\,\Big(\ C[\,\{M_1\}_k,\ldots,\{M_n\}_k\,]\ \mid \\ & \qquad D[\,\mathsf{case}\,x\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\ Q\,]\ \Big) \quad = \\ & (\nu \widetilde{h},k)\,\Big(\ C[\,\{M_1\}_k,\ldots,\{M_n\}_k\,]\ \mid \\ & \qquad D[\,\textstyle\sum_{i=1}^{n}[x = \{M_i\}_k]\ Q[M_i/y]\,]\Big)\end{aligned}$$

If k does not occur in contexts $C[\,\cdot,\ldots,\cdot\,]$ and $D[\,\cdot\,]$ and $\forall\ i = 1,\ldots,n$ C does not bind names in M_i .

4.3. The Kerberos example

Specification. For the sake of readability, we will use in the sequel a few obvious notational shorthands. For example $a(\langle y,z\rangle).P$ stands for $a(x).\,\mathsf{pair}\ x\ \mathsf{of}\ \langle y,z\rangle\ \mathsf{in}\ P$, $a(\{M\}_k).P$ stands for $a(x).\,\mathsf{case}\,x\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\,[y = M]P$, and $a(\{M,N\}_k).P$ stands for $a(x).\,\mathsf{case}\,x\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\ \ \mathsf{pair}\ y\ \mathsf{of}\ \langle z,t\rangle\ \mathsf{in}\ [z = M, t = N]P$.

Table 3 gives a high level specification of the protocol using these abbreviations, while Table 4 gives its translation into the syntax of Table 1. All bound names in K are assumed to be distinct from one another and from the free names. Subscripts should help reminding the expected value of each input variable. For instance, the expected value for x_{certB} is B's certificate, i.e. $\{T,k_{AB},A\}_{k_{BS}}$. Names A and B present in K refer the identity of the principals involved; names *inA* and *reB* are symbolic names that refer the processes associated to A and B respectively (i.e. the principal named A is the initiator of the protocol, while the principal named B is the responder). We decided to keep these names different in order to better distinguish between the principals and the code implementing them.

When starting the protocol execution, all the principals implicitly synchronize on the current time T ($\overline{clock}\langle T\rangle$). This is an approximation of what happens, as the spi-calculus does not provide explicit timing constructs implementing secure clock synchronization (a difficult task which may require complex interactions). Note that *reB* checks the presence of the timestamp T in the first received message and rejects any message not containing T.

Table 3
The Kerberos protocol in spi-calculus

$$
\begin{array}{rcl}
inA & \stackrel{\text{def}}{=} & \overline{c_{AS}}\langle A,B\rangle.\, c_{AS}(\{T,x_{k_{AB}},B,x_{cert_B}\}_{k_{AS}}).\overline{c_{AB}}\langle x_{cert_B},\{A,n_A\}_{x_{k_{AB}}}\rangle.\, c_{AB}(\{n_A\}_{x_{k_{AB}}}).\,\overline{commit_A}\langle\rangle.\,\mathbf{0}\\
reB & \stackrel{\text{def}}{=} & c_{AB}(\{T,y_{k_{AB}},A\}_{k_{BS}},\{A,y_{n_A}\}_{y_{k_{AB}}}).\,\overline{c_{AB}}\langle\{y_{n_A}\}_{y_{k_{AB}}}\rangle.\,\overline{commit_B}\langle\rangle.\,\mathbf{0}\\
S & \stackrel{\text{def}}{=} & c_{AS}(A,B).\,\overline{c_{AS}}\langle\{T,k_{AB},B,\{T,k_{AB},A\}_{k_{BS}}\}_{k_{AS}}\rangle.\,\mathbf{0}\\
L & \stackrel{\text{def}}{=} & \overline{lost}\langle\{T_{\mathtt{old}},k_{\mathtt{old}},A\}_{k_{BS}},k_{\mathtt{old}}\rangle.\,\mathbf{0}\\
C & \stackrel{\text{def}}{=} & \overline{clock}\langle T\rangle.\,\mathbf{0}\\
K & \stackrel{\text{def}}{=} & (\nu\, k_{AS},k_{BS})\Big(\ L\ \mid\ (\nu\, T)\Big(\ C\ \mid((\nu\, n_A)inA)\mid reB\mid((\nu\, k_{AB})S)\ \Big)\ \Big)
\end{array}
$$

Table 4
Full details of *inA*, *reB* and *S* for the Kerberos protocol

$$
\begin{array}{rcl}
inA & \stackrel{\text{def}}{=} & \overline{c_{AS}}\langle A,B\rangle.\, c_{AS}(x_1).\,\mathsf{case}\, x_1\ \mathsf{of}\ \{x_1'\}_{k_{AS}}\ \mathsf{in}\ \mathsf{pair}\ x_1'\ \mathsf{of}\ \langle x_T,x_{k_{AB}},x_B,x_{cert_B}\rangle\ \mathsf{in}\\
 & & [x_T=T,x_B=B]\ \overline{c_{AB}}\langle x_{cert_B},\{A,n_A\}_{x_{k_{AB}}}\rangle.\, c_{AB}(x_2).\,[x_2=\{n_A\}_{x_{k_{AB}}}]\,\overline{commit_A}\langle\rangle.\,\mathbf{0}\\
reB & \stackrel{\text{def}}{=} & c_{AB}(y).\,\mathsf{pair}\ y\ \mathsf{of}\ \langle y_1,y_2\rangle\ \mathsf{in}\ \ \mathsf{case} y_1\ \mathsf{of}\ \{y_1'\}_{k_{BS}}\ \mathsf{in}\ \mathsf{pair}\ y_1'\ \mathsf{of}\ \langle y_T,y_{k_{AB}},y_A\rangle\ \mathsf{in}\\
 & & [y_T=T,y_A=A]\ \mathsf{case} y_2\ \mathsf{of}\ \{y_2'\}_{y_{k_{AB}}}\ \mathsf{in}\ \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A',y_{n_A}'\rangle\ \mathsf{in}\\
 & & [y_A'=A]\overline{c_{AB}}\langle\{y_{n_A}'\}_{y_{k_{AB}}}\rangle.\,\overline{commit_B}\langle\rangle.\,\mathbf{0}\\
S & \stackrel{\text{def}}{=} & c_{AS}(z).\,\mathsf{pair}\ z\ \mathsf{of}\ \langle z_A,z_B\rangle\ \mathsf{in}\ \ [z_A=A,z_B=B]\overline{c_{AS}}\langle\{T,k_{AB},B,\{T,k_{AB},A\}_{k_{BS}}\}_{k_{AS}}\rangle.\,\mathbf{0}
\end{array}
$$

Outputs at channels $commit_A$ and $commit_B$ are used to signal that *inA* and *reB* have completed successfully the protocol. For readability, we have omitted the messages carried by these two actions, which are irrelevant here. The $\overline{lost}$-output action accounts for the accidental loss of an old session key $k_{\mathtt{old}}$ and of the corresponding certificate for B, $\{T_{\mathtt{old}},k_{\mathtt{old}},A\}_{k_{BS}}$.

Intuitively, everything works well because the long term keys k_{AS} and k_{BS} remain secret. Of course, if an intruder could forge e.g. k_{BS}, it would be possible for him to create a new certificate (with the current timestamp but with a non-authentic key) and it would be impossible for B to detect the event. Note that the system is not specified so as to guarantee that a commit will eventually be reached: we are only interested in checking that no "wrong" commit will ever happen.

Verification. We will consider authentication of the session key: *"B and A only accept the key k_{AB} generated by S"*. Formally, we want to prove that

$$(\ \varepsilon_I\ ,\ \varepsilon_I\ )\vdash K = K_{aut},$$

where ε_I denotes the environment that acts like the identity on the set of names $I \stackrel{\text{def}}{=} fn(K,K_{aut}) = \{clock,\ lost,\ c_{AS},\ c_{BS},\ c_{AB},\ commit_A,\ commit_B,\ A,\ B,\ T_{\mathtt{old}},\ k_{\mathtt{old}}\}$ and K_{aut}, defined below, formalises the desired protocol's behaviour. inA_{aut} and reB_{aut} can commit only upon receipt of the expected k_{AB} generated by S; in fact, note that K_{aut} is obtained from K's definition by adding the matchings $[x_{k_{AB}}=k_{AB}]$ and $[y_{k_{AB}}=k_{AB}]$ in *inA* and *reB* respectively, upon reception of their certificates.

$$
\begin{array}{rcl}
inA_{aut} & \stackrel{\text{def}}{=} & \overline{c_{AS}}\langle A,B\rangle.\, c_{AS}(\{T,k_{AB},B,x_{cert_B}\}_{k_{AS}}).\\
 & & \overline{c_{AB}}\langle x_{cert_B},\{A,n_A\}_{k_{AB}}\rangle.\, c_{AB}(\{n_A\}_{k_{AB}}).\\
 & & \overline{commit_A}\langle\rangle.\,\mathbf{0}\\
reB_{aut} & \stackrel{\text{def}}{=} & c_{AB}(\{T,k_{AB},A\}_{k_{BS}},\{A,y_{n_A}\}_{k_{AB}}).\\
 & & \overline{c_{AB}}\langle\{y_{n_A}\}_{k_{AB}}\rangle.\,\overline{commit_B}\langle\rangle.\,\mathbf{0}\\
K_{aut} & \stackrel{\text{def}}{=} & (\nu\, k_{AS},k_{BS},k_{AB})\Big(\\
 & & \quad L\ \mid(\nu\, T)\Big(\ C\ \mid((\nu\, n_A)inA_{aut})\mid\\
 & & \quad reB_{aut}\mid S\ \Big)\ \Big)
\end{array}
$$

We will prove the desired equality by applying the laws of Section 4.2. The proof consists of three steps:

(i) By (EXTR), $(\ \varepsilon_I\ ,\ \varepsilon_I\ )\vdash K = (\nu\, k_{AS},\ k_{BS},\ k_{AB},\ n_A,\ T)(L\mid C\mid inA\mid S\mid reB)$. By (CASE) applied to $\mathsf{case}\ x_1\ \mathsf{of}\ \ldots$ in *inA*, then by structural equivalence (axiom for pair splitting) and finally by (TRANS), we obtain $(\ \varepsilon_I\ ,\ \varepsilon_I\ )\vdash K = (\nu\, k_{AS},\ k_{BS},\ k_{AB},\ n_A,\ T)(L\mid C\mid inA'\mid S\mid reB)$, where

$$
\begin{array}{rcl}
inA' & \stackrel{\text{def}}{=} & \overline{c_{AS}}\langle A,B\rangle.\, c_{AS}(x_1).\\
 & & [x_1=\{T,k_{AB},B,\{T,k_{AB},A\}_{k_{BS}}\}_{k_{AS}}]\\
 & & \overline{c_{AB}}\langle\{T,k_{AB},A\}_{k_{BS}}\rangle.\,\{A,n_A\}_{k_{AB}}\\
 & & c_{AB}(x_2).\,[x_2=\{n_A\}_{k_{AB}}]\,\overline{commit_A}\langle\rangle.\,\mathbf{0}\ .
\end{array}
$$

(By (MATCH), we have deleted the tautological matchings $[T = T, B = B]$). We now apply (CASE) to case y_1 of ... in reB and similarly we obtain $(\varepsilon_I, \varepsilon_I) \vdash K = (\nu k_{AS}, k_{BS}, k_{AB}, n_A, T)(L \mid C \mid inA' \mid S \mid reB_1)$

where

$$
\begin{aligned}
reB_1 \stackrel{\text{def}}{=}\ & c_{AB}(y).\mathsf{pair}\ y\ \mathsf{of}\ \langle y_1, y_2\rangle\ \mathsf{in}\ \big(\\
& [y_1 = \{T, k_{AB}, A\}_{k_{BS}}, T = T, A = A] \\
& \mathsf{case} y_2\ \mathsf{of}\ \{y_2'\}_{k_{AB}}\ \mathsf{in}\ \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A', y_{n_A}'\rangle\ \mathsf{in} \\
& [y_A' = A]\ \overline{c_{AB}}\langle\{y_{n_A}'\}_{k_{AB}}\rangle.\overline{commit_B}\langle\rangle.\mathbf{0}\ + \\
& [y_1 = \{T_{\mathtt{old}}, k_{\mathtt{old}}, A\}_{k_{BS}}, T_{\mathtt{old}} = T, A = A] \\
& \mathsf{case} y_2\ \mathsf{of}\ \{y_2'\}_{k_{\mathtt{old}}}\ \mathsf{in}\ \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A', y_{n_A}'\rangle\ \mathsf{in} \\
& [y_A' = A]\ \overline{c_{AB}}\langle\{y_{n_A}'\}_{k_{\mathtt{old}}}\rangle.\overline{commit_B}\langle\rangle.\mathbf{0}\ \big)
\end{aligned}
$$

By (MATCH), we can delete the tautological matchings $[T = T, A = A]$ from the first summand and delete the second summand (the latter is stuck because of the failure of the matching between T and $T_{\mathtt{old}}$). Hence, by (TRANS), we have

$$
\begin{aligned}
(\varepsilon_I, \varepsilon_I) \vdash K \quad = \quad & (\nu k_{AS}, k_{BS}, k_{AB}, n_A, T) \\
& (L \mid C \mid inA' \mid reB') \qquad (1)
\end{aligned}
$$

where

$$
\begin{aligned}
reB' \stackrel{\text{def}}{=}\ & c_{AB}(y).\mathsf{pair}\ y\ \mathsf{of}\ \langle y_1, y_2\rangle\ \mathsf{in} \\
& [y_1 = \{T, k_{AB}, A\}_{k_{BS}}] \\
& \mathsf{case} y_2\ \mathsf{of}\ \{y_2'\}_{k_{AB}}\ \mathsf{in} \\
& \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A', y_{n_A}'\rangle\ \mathsf{in}\ [y_A' = A] \\
& \overline{c_{AB}}\langle\{y_{n_A}'\}_{k_{AB}}\rangle.\overline{commit_B}\langle\rangle.\mathbf{0}.
\end{aligned}
$$

(ii) Similarly, $(\varepsilon_I, \varepsilon_I) \vdash K_{aut} = (\nu k_{AS}, k_{BS}, k_{AB}, n_A, T)(L \mid C \mid inA' \mid S \mid reB_{aut})$. Then, applying (CASE) to case y_1 of ... in reB_{aut}, we obtain $(\varepsilon_I, \varepsilon_I) \vdash K_{aut} = (\nu k_{AS}, k_{BS}, k_{AB}, n_A, T)(L \mid C \mid inA' \mid S \mid reB'_{aut})$ where

$$
\begin{aligned}
reB'_{aut} \stackrel{\text{def}}{=}\ & c_{AB}(y).\mathsf{pair}\ y\ \mathsf{of}\ \langle y_1, y_2\rangle\ \mathsf{in}\ \big(\\
& [y_1 = \{T, k_{AB}, A\}_{k_{BS}}, T = T, A = A] \\
& [k_{AB} = k_{AB}]\ \mathsf{case} y_2\ \mathsf{of}\ \{y_2'\}_{k_{AB}}\ \mathsf{in} \\
& \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A', y_{n_A}'\rangle\ \mathsf{in}\ [y_A' = A] \\
& \overline{c_{AB}}\langle\{y_{n_A}'\}_{k_{AB}}\rangle.\overline{commit_B}\langle\rangle.\mathbf{0}\ + \\
& [y_1 = \{T_{\mathtt{old}}, k_{\mathtt{old}}, A\}_{k_{BS}}, T_{\mathtt{old}} = T, A = A] \\
& [k_{\mathtt{old}} = k_{AB}]\ \mathsf{case} y_2\ \mathsf{of}\ \{y_2'\}_{k_{\mathtt{old}}}\ \mathsf{in} \\
& \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A', y_{n_A}'\rangle\ \mathsf{in}\ [y_A' = A] \\
& \overline{c_{AB}}\langle\{y_{n_A}'\}_{k_{\mathtt{old}}}\rangle.\overline{commit_B}\langle\rangle.\mathbf{0}\ \big)
\end{aligned}
$$

Again by (MATCH), we can delete the tautological matchings from the first summand and delete the second one, obtaining

$$
\begin{aligned}
(\varepsilon_I, \varepsilon_I) \vdash K_{aut} \quad = \quad & (\nu k_{AS}, k_{BS}, k_{AB}, n_A, T) \\
& (L \mid C \mid inA' \mid S \mid reB') \qquad (2)
\end{aligned}
$$

(iii) The right hand sides of (1) and (2) are the same. Hence by (TRANS), we obtain the desired $(\varepsilon_I, \varepsilon_I) \vdash K = K_{aut}$.

Finally, notice that without the matching $[y_T = T]$ in reB's definition, the equivalence would be broken. In particular, upon receipt of $\{T_{\mathtt{old}}, k_{\mathtt{old}}, A\}_{k_{BS}}$, reB would perform a final $\overline{commit_B}$, which reB_{aut} cannot do. In essence, removing the check $[y_T = T]$ would recreate the well-known attack against the Needham-Schroeder protocol with symmetric encryption (see e.g. [12]).

5. Trace analysis

We outline here a verification method that departs from the concept of observational equivalence discussed in the previous section. The method is based on analysing the execution traces of a single process representing the protocol. Recall that a *trace* is a sequence of I/O events (actions) executable by a given spi-calculus process. Roughly, a sensible way of expressing authentication of A towards B, in our version of Kerberos, is requiring that, in every trace generated by K, B's final input action is preceded by an A's output of the same message, i.e. B will only accept messages originating from A (similarly for authentication in the other direction).

Trace-based formalizations of authentication and secrecy are generally less demanding than equivalence-based formulations, but more amenable to automatic checking. We will say more on pros and cons of the two approaches in Section 6.

A crucial aspect of the trace analysis method is a notion of *symbolic* execution [7] that avoids having to explicitly consider the infinitely many traces generated by the protocol. This form of state-explosion is related to the interaction of each participant with the external environment. Symbolic execution has been implemented as part of a prototype verification tool named STA (*Symbolic Trace Analyzer*) implemented in ML [8].

In the rest of the section we will first outline the model underlying trace analysis, then touch upon the method of symbolic execution and finally re-consider the Kerberos example in the light of trace analysis.

5.1. Overview of the model

The model underlying the trace analysis method is very close in spirit to Dolev-Yao's one [15]. Informally, agents executing the protocol communicate through a network of public channels that are under the control of an adversary,

therefore there are no private, secure channels. Sending a message just means handing the message to the adversary. Conversely, receiving a message just means accepting any message among those the adversary can produce. The adversary records all messages that transit over the network, and can produce a message by either replaying an old one, or by combining old messages (e.g. by pairing, encryption and decryption) and/or by generating fresh quantities.

Formally, a state of the system is a pair $s \triangleright P$, called *configuration*: s is a trace of past I/O events (actions), and represents the current adversary's knowledge; P is a spi-term, describing the intended behavior of honest participants. The set of all configurations is denoted by $\mathscr{C}$. The dynamics of configurations is given by a transition relation $\longrightarrow \subseteq \mathscr{C} \times \mathscr{C}$, that describes elementary steps of computations. In Table 5 we report the rules defining the transition relation, for a subset of the language introduced in Section 2. In particular, since we are looking for an automatic

Table 5
Transition relation on configurations ($\longrightarrow$)

(INP)	$s \triangleright \mathsf{a}(x).P \longrightarrow s \cdot \mathsf{a}\langle M\rangle \triangleright P[M/x]$
	if $s \vdash M$ and M is closed
(OUT)	$s \triangleright \overline{\mathsf{a}}\langle M\rangle.P \longrightarrow s \cdot \overline{\mathsf{a}}\langle M\rangle \triangleright P$
(CASE)	$s \triangleright \mathsf{case}\,\{M\}_k\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\,P \longrightarrow s \triangleright P[M/y]$
(SPLIT)	$s \triangleright \mathsf{pair}\ \langle M,N\rangle\ \mathsf{of}\ \langle x,y\rangle\ \mathsf{in}\ P \longrightarrow \longrightarrow s \triangleright P[M/x, N/y]$
(MATCH)	$s \triangleright [M=M]P \longrightarrow s \triangleright P$
(RES)	$s \triangleright (\nu a)P \longrightarrow s[a'/a] \triangleright P[a'/a]$
	if a' is fresh for s
(PAR)	$\dfrac{s \triangleright P \longrightarrow s' \triangleright P'}{s \triangleright P \mid Q \longrightarrow s' \triangleright P' \mid Q}$
plus symmetric version of (PAR).	

method, we have omitted replication, which would make the problem undecidable (see e.g. [16]). Rules (INP) and (OUT) concern sending and receiving messages, respectively. Since sending a message just means handing the message to the adversary, any output action $\overline{\mathsf{a}}\langle M\rangle$ fired by a process is recorded in the adversary's current knowledge s (rule (OUT)). Conversely, receiving a message just means accepting any message among those the adversary can produce. Therefore, in rule (INP) the variable x can be replaced by any message M non-deterministically chosen among those the adversary can synthesize from its current knowledge s. The synthesis of a message M from a set of known messages S is formalized by a deduction relation $\vdash$. Here is a sample of deduction rules defining $\vdash$ (see [7]):

$$\frac{M \in S}{S \vdash M} \qquad \frac{S \vdash M \qquad S \vdash k}{S \vdash \{M\}_k}$$

$$\frac{S \vdash \{M\}_k \qquad S \vdash k}{S \vdash M}$$

The other operational rules in Table 5 govern how a process decrypts a message ($\mathsf{case}\,M\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\,A$), splits a pair ($\mathsf{pair}\ \langle M,N\rangle\ \mathsf{of}\ \langle x,y\rangle\ \mathsf{in}\ A$), compares two messages for equality ($[M=N]A$), handles a new name ($(\nu a)P$) and interleaves execution of parallel threads ($A \mid B$).

It is worthwhile to point out that there is no need for an explicit description of the adversary's behavior, as the latter is wholly determined by its current knowledge – the s in $s \triangleright P$ – and by the deduction relation $\vdash$. This is somehow in contrast with other proposals [21, 26], where the adversary must be explicitly described, but it is comform to [6, 18, 27].

Given a configuration $s \triangleright P$ and a trace s', we say that $s \triangleright P$ *generates* s' if $s \triangleright P \longrightarrow^* s' \triangleright P'$ for some P' ($\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$, i.e. zero or more steps of $\longrightarrow$). We express properties of the protocol in terms of the traces it generates. In particular, we focus on correspondence assertions of the kind:

> for every generated trace, if action β occurs in the trace, then action α must have occurred at some previous point in the trace,

that is concisely written as $\alpha \hookleftarrow \beta$. More accurately, we allow α and β to contain free variables, that may be instantiated to ground values. Thus $\alpha \hookleftarrow \beta$ actually means that *every instance* of β must be preceded by the corresponding instance of α, for every generated trace. We write $s \triangleright P \models \alpha \hookleftarrow \beta$ if the configuration $s \triangleright P$ satisfies this requirement. This kind of assertions is flexible enough to express interesting secrecy and authentication properties. As an example, the final step of many key-establishment protocols consists in A's sending a message of the form $\{N\}_k$ to B, where N is some authentication information, and k the newly established key. A typical property one wants to verify is that any message encrypted with k that is accepted by B at the final step should actually originate from A (this ensures B he is really talking to A, and that k is authentic). If we call $\mathsf{final}_\mathsf{A}$ and $\mathsf{final}_\mathsf{B}$ the labels attached to A's and B's final action, respectively, then the property might be expressed by $\overline{\mathsf{final}_\mathsf{A}}\langle\{x\}_k\rangle \hookleftarrow \mathsf{final}_\mathsf{B}\langle\{x\}_k\rangle$, for x a variable. The scheme also permits expressing secrecy as a reachability property (in the style of [5, 18]): this is further discussed in Section 6.

5.2. Symbolic execution

When synthesizing new messages, the adversary can apply operations like pairing, encryption and generation of fresh names, an arbitrary number of times. Thus the set of messages the adversary can synthesize at any time is actually infinite in general (i.e. if not empty). Any such message can be non-deterministically chosen by the adversary and sent to a participant willing to receive it; therefore every model based on Dolev and Yao's is in principle infinite. Our model makes no exception: in rule (INP) the set of M s.t. $s \vdash M$ is always infinite, and this makes the model infinitely-branching. This can be regarded as a state explosion problem induced by message exchange.

To overcome this problem, the STA tool implements a verification method based on a notion of symbolic execution. A new transition relation (written $\longrightarrow_s$, below) is introduced in order to condense the infinitely many transitions that arise from an input action (rule (INP) in Table 5) into a single, *symbolic* transition. The received message is now represented simply by a free variable, whose set of possible values is constrained as the execution proceeds. Technically, a constraint takes the form of *most general unifier* (mgu), i.e., the most general substitution that makes two expressions equal. The set of traces generated using the symbolic transition relation constitutes the *symbolic model* of the protocol. Differently from the standard model given by $\longrightarrow$, the symbolic model is finite, because each input action just gives rise to one symbolic transition and agents cannot loop.

For a flavor of how symbolic execution works, let us consider an example focusing on shared-key encryption. Suppose that agent P, after receiving a message, tries decrypting this message using key k; if this succeeds and y is the result, the agent checks whether y equals b and, if so, proceeds like P'. This is written as $P \stackrel{\text{def}}{=} \mathsf{a}(x).\,\mathsf{case}\,x\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\,[y=b]P'$, for y fresh. Let us explain how the symbolic execution proceeds, starting from the initial configuration $\varepsilon \triangleright P$. After the first input step, in the second step the decryption $\mathsf{case}\,x\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\cdots$ is resolved by unifying x and $\{y\}_k$, which results in the substitution $[\{y\}_k/x]$. In the third step, the equality test $[y=b]$ is in turn resolved by unifying y and b, that results in $[b/y]$. Formally,

$$\begin{array}{rll} \varepsilon \triangleright P & \longrightarrow_s & \mathsf{a}\langle x\rangle \triangleright \mathsf{case}\,x\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\,[y=b]P' \\ & \longrightarrow_s & \mathsf{a}\langle\{y\}_k\rangle \triangleright [y=b]P'[\{y\}_k/x] \\ & \longrightarrow_s & \mathsf{a}\langle\{b\}_k\rangle \triangleright P'[\{y\}_k/x][b/y]. \end{array}$$

An important point is that symbolic execution actually ignores the deduction relation $\vdash$ and thus may give rise to "inconsistent" symbolic traces. These inconsistencies can be detected and discovered via a refinement procedure described in [7].

The verification method based on symbolic execution is proven sound and complete w.r.t. the standard model, in the sense that every consistent attack detected in the symbolic model (relation $\longrightarrow_s$) corresponds to some attack in the standard model (relation $\longrightarrow$), and vice-versa. In other words, the symbolic model captures all and only the attacks of the standard model. For instance, the method detects type-dependent attacks, which usually escape finite-state analysis, e.g. [22]. In this kind of attacks, the adversary cheats on the type of some messages, e.g. by inserting a nonce where a key is expected according to the protocol description.

5.3. The Kerberos example

We illustrate the trace analysis method and the use of the automatic tool STA on the simplified Kerberos protocol of Section 3. The tool follows the syntax and semantics of the formal model, with a few minor differences. E.g., action prefixing is written `>>`, parallel composition is written `||`, restriction is written `new-in`, while **0** is written `stop`. Output actions are written as `a!M`, while input actions are written as `a?M`. Note that `M` can be a generic message pattern: this means receiving any adversary-generated message whose form matches `M`. To this purpose, we distinguish explicitly between names and variables (the latters, by convention, start by `x`, `y`, ...). Finally, with `<--` we mean the predicate $\hookleftarrow$ and with `[] @ K` the configuration $\varepsilon \triangleright K$. What follows is the complete STA script defining one session of Kerberos, and the desired authentication properties. Since all channels are public and controlled by the environment, we have made all channel names distinct and used them as references for process actions. Also, we need not make commit actions explicit now, thus we have dropped them.

`Conf` is the initial configuration of the protocol, composed by an empty list of actions and by `K` while `AuthKey`, `AuthAtoB` and `AuthBtoA` represent the properties we want to check of this configuration. `AuthKey` states that any message accepted by A at `a2` should originate from S: this implies the adversary cannot fool A into accepting a key different from `kAB`. Property `AuthAtoB` states that any message accepted by B at `b1` should originate from A at `a3`. `AuthBtoA` can be explained similarly. The three properties together guarantee that A and B always talk to each other, and that they agree on the exchanged data (in particular, on the established key), which are authentic.

If we ask STA to check any of the three properties listed above, we get this answer:

```
> val it = "No attack was found, 61
  symbolic configurations reached."
      : string
```

which means that STA has explored the whole symbolic state-space of the protocol, consisting of 61 configurations, without finding any trace violating the property (this exploration takes STA a fraction of a second). Thus there are no attacks on this configuration of the protocol.

Suppose now we modify B so that it omits the check on the freshness of T, i.e. we re-define

```
val reB=b1?({t,ykAB,A}kBS,{A,ynA}KAB) >>
        b2!{ynA}ykAB >> stop;
```

where we have replaced the timestamp `T` by an arbitrary variable `t` in `b1`. STA finds an attack on the property

```
val inA      = nA new-in ( a1!(A,B) >> a2?{T,xkAB,B,xCertB}KAS >>
                  a3!(xCertB, {A,nA}xkAB) >> a4?{nA}xkAB >> stop );
val S        = kAB new-in ( s1?(A,B) >>
                  s2!{T,kAB,B,{T,kAB,A}kBS}kAS >> stop);
val reB      = b1?({T,ykAB,A}kBS,{A,ynA}ykAB) >>
                  b2!{ynA}ykAB >> stop;
val K        = kAS new-in kBS new-in ( lost!(kOld,{TOld,kOld,A}KBS)>>stop ||
                  T new-in ( clock!T >> stop || inA || reB || S ) );
val Conf     = ( [] @ K );
val AuthKey  = (s2!t <-- a2?t);
val AuthAtoB = (a3!u <-- b1?u);
val AuthBtoA = (b2!w <-- a4?w);
```

AuthAtoB. The attack is reported under the form of a trace violating the property:

```
>   val it = "An attack was found:
    lost!(kOld,{TOld,kOld,A}kBS).
    clock!T. a1!(A,B).
    b1?({TOld,kOld,A}kBS1,{A,ynA}kOld)
4   symbolic configurations
    reached." :  string
```

The attack is based on the adversary's replaying the old, compromised key kOld and the corresponding certificate {TOld,kOld,A}kBS acquired thanks to the lost action. Note that the trace contains a free variable ynA: it can take on any value which is known to the attacker.

6. A comparison of two methods

An important problem left open by current research is that of establishing a precise relationship between the notions of authentication and secrecy conveyed by the two models outlined in the previous sections.

The equivalence-based formalization is seemingly more demanding than the trace-based one. In fact, the former takes into account the overall behaviour of the protocol – including I/O traffic – while the latter takes into account only correspondence between single actions, or exposure of secret data items. Surprisingly, the two notions are formally incomparable: we show below that neither is stronger than the other. Thus, adopting one notion or the other is not a matter of relative strength. We shall confine our discussion to secrecy, but we feel that similar arguments apply in the case of authentication. First of all, let us state more precisely the notions of secrecy we are interested in.

Definition 3 (two notions of secrecy). Let $P(x)$ be a spi-calculus process. We say that:

- $P(x)$ *keeps x E-secret* if for every x': $P(x) \simeq P(x')$;
- $P(x)$ *keeps x T-secret* if there is no configuration $s' \triangleright P'$ s.t. $\varepsilon \triangleright P(x) \longrightarrow^* s' \triangleright P'$ and $s' \vdash x$.

Now, consider the process $P(x) \stackrel{\text{def}}{=} (\nu k)(a(y).[y = x]\overline{b}\langle\{x\}_k\rangle.\mathbf{0}$. The process $P(x)$ keeps x T-secret (by inspection), but not E-secret. In fact, consider the observer $O \stackrel{\text{def}}{=} \overline{a}\langle x\rangle.b(z).\omega.\mathbf{0}$: we have $P(x) \mid O \stackrel{\omega}{\Longrightarrow}$, but *not* $P(x') \mid O \stackrel{\omega}{\Longrightarrow}$, hence $P(x) \not\simeq P(x')$ for $x' \neq x$.

On the other hand, consider $Q(x) \stackrel{\text{def}}{=} a(y).([y = x]\overline{b}\langle x\rangle.\mathbf{0} \mid !\overline{b}\langle y\rangle.\mathbf{0})$. Clearly, $Q(x)$ does *not* keep x T-secret. However, $Q(x)$ and $Q(x')$ are trace-equivalent, hence testing equivalent, for any x'; this is a consequence of the fact that $\overline{b}\langle x\rangle.\mathbf{0} \mid !\overline{b}\langle x\rangle.\mathbf{0} \equiv !\overline{b}\langle x\rangle.\mathbf{0}$.

The above examples show that E-secrecy does not imply T-secrecy, and, conversely, that T-secrecy does not imply E-secrecy.

7. Concluding remarks and related work

We have outlined some recent approaches to the analysis of security protocols, centered around concepts derived from the field of process calculi, such as observational semantics and symbolic transition systems.

Early work on reasoning methods for the spi-calculus was presented in [4], where *framed bisimulation* was introduced as a proof technique, though incomplete, for reasoning on contextual equivalences. The environment sensitive transition system presented here was introduced in [10], and based on that, the complete characterizations of contextual semantics discussed in Section 4 were obtained. Some of the reasoning principles used in this paper were introduced there. A sound and complete proof system is discussed in [11].

Concerning trace analysis, [7] develops the theory underlying the verification tool STA, while [8] presents verification examples and compares the results to those obtained using finite-state methods. Initial work on symbolic analysis is due to Huima [19]. Symbolic techniques are also exploited in [5, 13, 29], but the algorithms they use are quite different from ours.

Another possible approach consists in deriving properties via type systems: example of these techniques are the type systems in [1, 2] for secrecy and in [17] for authentication. When compared to more traditional methods – like CSP-based model checking [21, 26] – major benefits of the

equivalence-based approach seem to be a host of syntax-driven reasoning principles and a fully satisfactory formalization of many important properties, including implicit information flow (that may arise due, e.g., to traffic analysis). On the other hand, the equivalence-based method lacks at present automatic verification techniques. Symbolic trace analysis appears to be closer in spirit to model checking, but does not suffer from the state-explosion problems of model checking, which requires considering approximate models, even when the number of protocol sessions is bounded. Moreover, finite-state model checking has proven very effective in practice to find bugs in security protocols, e.g. [22]. Analysis of real-life case-studies could tell whether the approaches derived from the spi-calculus may represent a valid alternative to the established techniques.

Acknowledgements

We are very grateful to the editor for careful reading and useful suggestions.
This work has been partially supported by EU within the FET – Global Computing initiative, project MIKADO IST-2001-32222 and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.

References

[1] M. Abadi, "Secrecy by typing in security protocols", *J. ACM*, vol. 46, no. 5, pp. 749–786, 1999.

[2] M. Abadi and B. Blanchet, "Analyzing security protocols with secrecy types and logic programs", in *POPL'02*. ACM Press, 2002.

[3] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: the spi-calculus", *Inform. Comput.*, vol. 148, no. 1, pp. 1–70, 1999.

[4] M. Abadi and A. D. Gordon, "A bisimulation method for cryptographic protocols", *Nord. J. Comput.*, vol. 5, no. 4, pp. 267–303, 1998.

[5] R. M. Amadio and S. Lugiez, "On the reachability problem in cryptographic protocols", in *Proc. CONCUR'00, LNCS*. Springer, 2000, vol. 1877 (full version: RR 3915, INRIA Sophia Antipolis).

[6] D. Bolignano, "Towards a mechanization of cryptographic protocol verification", in *International Conference on Computer Aided Verification, LNCS*. Springer, 1997.

[7] M. Boreale, "Symbolic trace analysis of cryptographic protocols", in *ICALP'01, LNCS*. Springer, 2001, vol. 2076, pp. 667–681.

[8] M. Boreale and M. G. Buscemi, "Experimenting with STA, a tool for automatic analysis of security protocols", in *ACM Symposium on Applied Computing 2002*. ACM Press, 2002.

[9] M. Boreale and R. De Nicola, "Testing equivalence for mobile processes", *Inform. Comput.*, vol. 120, pp. 279–303, 1995.

[10] M. Boreale, R. De Nicola, and R. Pugliese, "Proof techniques for cryptographic processes", in *LICS'99, Proceedings*. IEEE Computer Society Press, 1999, pp. 157–166 (full version to appear in *SIAM J. Comput.*).

[11] M. Boreale and D. Gorla, "On compositional reasoning in the spi-calculus", in *FoSSaCS'02, Proceedings*, M. Nielsen and H. U. Engberg, Eds., *LNCS*. Springer, 2000, vol. 2303, pp. 67–81.

[12] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication", *ACM Trans. Comput. Syst.*, vol. 8, no. 1, pp. 18–36, 1990.

[13] H. Comon, V. Cortier, and J. Mitchell, "Tree automata with one memory, set constraints and ping-pong protocols", in *ICALP'01, LNCS*. Springer, 2001, vol. 2076, pp. 682–693.

[14] R. De Nicola and M. C. B. Hennessy, "Testing equivalence for processes", *Theor. Comput. Sci.*, no. 34, pp. 83–133, 1984.

[15] D. Dolev and A. Yao, "On the security of public-key protocols", *IEEE Trans. Inform. Theory*, vol. 2, no. 29, pp. 198–208, 1983.

[16] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov, "Undecidability of bounded security protocols", in *Proc. FLOC Worksh. Form. Meth. Secur. Protoc.*, Trento, Italy, 1999.

[17] A. D. Gordon and A. Jeffrey, "Authenticity by typing for security protocols", in *14th IEEE Comput. Secur. Found. Worksh.*, 2001, pp. 145–159.

[18] J. Goubault-Larrecq, "A method for automatic cryptographic protocol verification", in *Proc. 15th IPDPS Workshops, LNCS*. Springer, 2000, vol. 1800, pp. 977–984.

[19] A. Huima, "Efficient infinite-state analysis of security protocols", in *Proc. FLOC Worksh. Form. Meth. Secur. Protoc.*, Trento, Italy, 1999.

[20] J. Kohl and B. Neuman, "The Kerberos network authentication service (version 5)". Internet Request for Comment RFC-1510, 1993.

[21] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR", in *TACAS'96*, *Proceedings*, T. Margaria and B. Steffen, Eds., *LNCS*. Springer, 1996, vol. 1055, pp. 147–166.

[22] G. Lowe, "A hierarchy of authentication specifications", in *Proc. 10th IEEE Computer of Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[23] R. Milner, "The polyadic π-calculus: a tutorial", in *Logic and Algebra of Specification*, F. L. Hamer, W. Brauer, and H. Schwichtenberg, Eds. Springer, 1993.

[24] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes (Part I and II)", *Inform. Comput.*, vol. 100, pp. 1–77, 1992.

[25] R. Milner and D. Sangiorgi, "Barbed bisimulation", in *ICALP'92*, *Proceedings*, W. Kuich, Ed., *LNCS*. Springer, 1992, vol. 623, pp. 685–695.

[26] J. C. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using Murφ", in *Proceedings of Symposium Security and Privacy*. IEEE Computer Society Press, 1997.

[27] L. C. Paulson, "The inductive approach to verifying cryptographic protocols", *J. Comput. Secur.*, no. 6, pp. 85–128, 1998.

[28] D. Pointcheval, "Asymmetric cryptography and practical security", *J. Telecommun. Inform. Technol.*, no. 4, pp. 41–56, 2002.

[29] M. Rusinowitch and M Turuani, "Protocol insecurity with finite number of sessions in NP-complete", in *14th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2001.

Michele Boreale received a Laurea degree in Scienze dell'Informazione from the University of Pisa in 1991, and a Ph.D. degree in computer science from the University "La Sapienza", Rome, in 1995. He has been research associate at the Dipartimento di Scienze dell'Informazione of University "La Sapienza" from February 1996 to July 1999, when he moved to the Dipartimento di Sistemi e Informatica of the University of Florence. His research interests include formal methods for specifying and verifying concurrent and reactive systems; process calculi and behavioral equivalences, particularly from the angle of elucidating their expressive power and finding tractable proof methods.
e-mail: boreale@dsi.unifi.it
Dipartimento di Sistemi e Informatica
Università di Firenze
Firenze, Italy

Daniele Gorla was born in Rome in 1976; he received his master degree "cum laude" in computer science by the University of Rome "La Sapienza" in 2000. He is currently a Ph.D. student in Florence under the supervision of prof. Rocco De Nicola within the Concurrency and Mobility Group, and he collaborates with the University of Rome.

e-mail: gorla@dsi.uniroma1.it
University of Rome, Italy
e-mail: gorla@dsi.unifi.it
Dipartimento di Sistemi e Informatica
Università di Firenze
Firenze, Italy