Formal Proofs of Security Protocols
David Baelde, ENS Paris-Saclay

December 21, 2018

Contents

1 Model .................................................. 2
1.1 Terms ............................................. 2
1.1.1 Equational theory ............................. 3
1.1.2 Rewrite rules ................................... 3
1.1.3 Renaming ....................................... 4
1.2 Processes ........................................... 5
1.2.1 Internal reduction ............................. 6
1.2.2 Labelled transitions ......................... 7

2 Verifying secrecy for bounded executions ............. 11
2.1 Deciding deduction ............................... 11
2.2 Symbolic execution ............................... 12
2.3 Solving deducibility constraint systems ............ 14

3 Verifying secrecy for unbounded executions .......... 15

4 Equivalences ......................................... 15

5 Verifying equivalences ................................ 15

6 Advanced topics ...................................... 15
We present in section 1 a formal model for representing security protocols, and illustrate how it can be used, e.g. to model secrecy. Secrecy verification is developed in more detail in sections 2 and 3 respectively for bounded and unbounded executions: in the former case we obtain a decidability result by using deducibility constraints; in the latter we describe the semi-decision procedure behind Proverif, using a Horn-clause abstraction of protocols. We go back to the semantics of our processes in section 4 to define and study various behavioral equivalences, and show how they are useful to model more advanced security properties such as strong secrecy, anonymity, unlinkability, etc. Section 5 is dedicated to the automated verification of equivalences. We will conclude in section 6 with yet undetermined advanced topics.

1 Model

Before anything else, we must define a formal model for security protocols. As is common, we shall use a variant of the (applied) π-calculus. More specifically, the calculus defined below is very close to the one used in Proverif.

The first step is to define a term language to represent messages and computations over them (section 1.1). Then, a process calculus will be used to represent protocols (section 1.2).

1.1 Terms

Terms are formal representations of messages and computations over them. We start by assuming several disjoint and countable sets of basic objects:

- a set $X$ of variables, which will be denoted by $x, y, z$;
- a set $N$ of names, which will be denoted by $n, m, k$.

Then, we assume a signature $\Sigma$, that is a set of function symbols together with an arity $ar_{\Sigma}: \Sigma \rightarrow \mathbb{N}$. Given a set of basic terms $B$, the set $T(B)$ of terms generated from $B$ using $\Sigma$ is defined as the least set containing $B$ and closed by application of function symbols respecting their arities. Terms will be denoted by $s, t, u, v$.

Example 1. One possible signature is $\Sigma = \{ \text{senc}, \text{sdec}, \text{pair}, \text{proj}_1, \text{proj}_2, \text{ok} \}$. The symbols senc and sdec, of arity 2, represent symmetric encryption and decryption. Pairing is modeled using pair of arity 2 and projection functions proj$_1$ and proj$_2$, both of arity 1. Finally, the symbol ok is of arity 0, i.e. it is a constant. With this signature we have pair(ok, ok) $\in T(\emptyset)$, senc(pair(ok, ok), k) $\in T(N)$ and sdec(ok, x) $\in T(X)$. However, senc(ok) and ok(ok) are not terms.

When using this signature, we will often write $(s, t)$ rather than pair$(s, t)$, and $\{m\}_k$ for senc$(m, k)$.

Given a term $t$, we define $fv(t)$ as the set of variables that occur in $t$. Similarly, $fn(t)$ is the set of names occurring in $t$. A term is said to be closed when it contains no variable. A substitution is a finite domain map from $X$ to $T(B)$ for some $B$. Substitutions will be denoted by $\theta$ or $\sigma$, their domain will be noted $\text{dom}(\cdot)$. A substitution $\{x_i \mapsto t_i\}_{i \in [1:n]}$ may also be written $[t_i/x_i]_{i \in [1:n]}$. In particular, $[t/x]$ is the substitution of domain $\{x\}$ that maps $x$ to $t$. The application of a substitution $\theta$ to a term $t$ is defined as usual and noted $t\theta$. In particular, when $\text{dom}(\theta) \cap fv(t) = \emptyset$, $t\theta = t$.

We will introduce two mechanisms for giving a meaning to function symbols. First, we will introduce an equations to model when two terms should be considered as being equal, i.e. when they represent computations that yield the same result. For
example, we may equate \( \text{proj}_j(\text{pair}(s, t)) \) and \( s \). Second, we provide our term algebra with a means to describe computations that may fail. For example, we may have that \( \text{sdec}(\text{senc}(s, k), k) \) reduces to \( s \) but \( \text{sdec}(\text{ok}, k) \) fails, indicating an encryption scheme where it is possible to distinguish random messages from actual ciphertexts.

Equations and reductions will be separate mechanisms, each one taking place on a specific kind of function symbol. Before introducing them, we thus assume that our signature is split between \( \text{constructor} \) and \( \text{destructor} \) symbols, i.e. \( \Sigma = \Sigma_c \cup \Sigma_d \). In the following we write \( T_c(B) \) for terms built from \( B \) using only constructor symbols, i.e., elements of \( \Sigma_c \). Elements of \( T_c(N) \) are called messages.

1.1.1 Equational theory

Our equational theory is going to be generated from equations between terms that may contain variables but no names. We thus assume a set of equations \( E \subseteq T_c(\mathcal{X}) \); we will use an infix notation for it, writing \( s \ E t \) rather than \( (s, t) \in E \). Then, for any set of basic terms \( B \) we define the binary relation \( \equiv_B \) over \( T(B \cup \mathcal{X}) \) as the least equivalence relation that contains \( E \) and is closed under substitution and context closure. In other words, we impose that:

- for all \( s \in B \), we have \( s =^E t \);
- for all \( s =^E t \) and for any substitution \( \theta : \mathcal{X} \to T(B \cup \mathcal{X}) \), we have \( s\theta =^E t\theta \);
- for all \( f \in \Sigma \) with \( \text{ar}(f) = n \),
  - for all \( s_1, \ldots, s_n \) and \( t_1, \ldots, t_n \) such that \( s_i =^E t_i \) for all \( i \in [1; n] \),
  - we have \( f(s_1, \ldots, s_n) =^E f(t_1, \ldots, t_n) \).

**Example 2.** With the signature of \( \text{ex} \) and assuming that \( \Sigma = \Sigma_c \), consider \( E \) made of three equations:

\[ \text{sdec}(\text{senc}(x, y), y) \ E x, \quad \text{and} \quad \text{proj}_j(\text{pair}(x_1, x_2)) \ E x_i \quad \text{for} \ i \in \{1, 2\}. \]

We then have \( \text{proj}_1(\text{sdec}(\text{senc}(\text{pair}(\text{ok}, n), k), k)) =^E \text{ok but} \text{ok} \not=^E \text{pair}(\text{ok}, n) \).

**Exercise 1.** Assume \( B \subseteq C \). Using the induction principle associated to \( =_E \), show that, for any \( s, t \in T(B \cup \mathcal{X}) \), \( s =^E t \) implies \( s =^E t \).

The previous exercise justifies that we do not need to specify \( B \) explicitly when considering an equality: we will simply write \( s =_E t \), meaning that \( s =^E t \) holds for any \( B \) such that \( s, t \in T(B \cup \mathcal{X}) \).

**Exercise 2.** Let \( u \) and \( v \) be terms such that \( u =_E v \), and let \( x \) be a variable. Show that \( t[u/x] =_E t[v/x] \) for any term \( t \). Conclude that the substitution principle holds: for any terms \( s, t, u, v \) and variable \( x \), \( s =_E t \) and \( u =_E v \) imply \( s[u/x] =_E t[v/x] \).

1.1.2 Rewrite rules

We assume, for each destructor symbol \( f \in \Sigma_d \) of arity \( n \), a set of \( \text{reduction rules} \) of the form \( f(u_1, \ldots, u_n) \to u \) where \( u, u_1, \ldots, u_n \in T_c(\mathcal{X}) \). From this we define a \( \text{computation relation} \downarrow \subseteq T(\mathcal{N}) \times T_c(\mathcal{N}) \) between terms and messages as the least relation satisfying the following conditions:

- for all \( n \in \mathcal{N}, n \downarrow n \);
- for all \( f \in \Sigma_c \) with \( \text{ar}(f) = n \),
  - for all \( t_1, \ldots, t_n \) and \( u_1, \ldots, u_n \) such that \( t_i \downarrow u_i \) for all \( i \in [1; n] \),
  - \( f(t_1, \ldots, t_n) \downarrow f(u_1, \ldots, u_n) \);
- for all \( f \in \Sigma_d \) with reduction rule \( f(u_1, \ldots, u_n) \to u \),
  - for all \( t_1, \ldots, t_n \) and \( \theta : \mathcal{X} \to T_c(\mathcal{N}) \) such that \( t_i \downarrow u_i \theta \) for each \( i \in [1; n] \),
  - \( f(t_1, \ldots, t_n) \downarrow u \theta \);

\( \downarrow \) should not be able to distinguish specific names, because names represent values that are generated at random.

As for the equational theory, specific applications of our model may call for extra assumptions. For instance, one may impose that the computation relation is deterministic (up to \( =_E \)) or computable.
• for all \( t, u \) and \( v \) such that \( t \Downarrow u \) and \( u =_E v \), \( t \Downarrow v \).

We write \( t \Downarrow \psi \) when there is no message \( u \) such that \( t \Downarrow u \).

**Example 3.** Consider again \( \Sigma \) from ex. \([7]\) but assuming now that \( \Sigma_d = \{ \text{proj}_1, \text{proj}_2 \} \). Assume that \( E \) contains only the equation \( \text{sdec}(\text{senc}(x, y), y) E x \), and take the reduction rules

\[
\text{proj}_1(\text{pair}(x, y)) \rightarrow x \quad \text{and} \quad \text{proj}_2(\text{pair}(x, y)) \rightarrow y.
\]

As an analogue of what we obtained in the previous example, we have

\[
\text{proj}_1(\text{sdec}(\text{senc}(\text{pair}(\text{ok}(n), n), k), k)) \Downarrow \text{ok}.
\]

Observe that the fourth item of the definition of \( \Downarrow \) is crucial to obtain this computation, as it is needed to have \( \text{sdec}(\text{senc}(\text{pair}(\text{ok}(n), n), k), k) \Downarrow \text{pair}(\text{ok}, n) \). We also have \( \text{pair}(\text{ok}, n) \Downarrow \text{ok} \): in fact, \( \text{pair}(\text{ok}, n) \Downarrow u \iff u =_E \text{pair}(\text{ok}, n) \). Finally, there are terms that cannot be computed, e.g. \( \text{proj}_1(\text{ok}) \Downarrow \) and \( \text{proj}_1((\text{ok}, \text{proj}_1(\text{ok}))) \Downarrow \).

Depending on the problem that is considered, it may be more practical to consider only equations and reduction rules. Sometimes, e.g. in Proverif, both are available. In such cases, there is often a choice between equations and reductions, as illustrated in the previous examples for pairing. This choice may involve performance issues but it also affects the adequacy of the modelling of cryptographic primitives. The key difference to keep in mind is that, when \( t \) is a term featuring destructors, it is sometimes impossible to obtain a message \( u \) such that \( t \Downarrow u \): this failure to compute (or failure to eliminate destructors) will lead to different behaviours of the protocol (and attacker). We will see below examples where it makes a difference.

**Example 4.** Asymmetric encryption is generally better represented as a destructor, using binary encryption and decryption symbols as well as a unary public-key symbol \( \text{pub} \), and the following reduction rule:

\[
\text{adec}(\text{aenc}(x, \text{pub}(y)), y) \rightarrow x
\]

**Example 5.** More expressive computations can be expressed by ordering the reduction rules associated to a destructor, and requiring that a rule may only be used if the previous ones do not apply. For instance, if \( \text{eq} \in \Sigma_d \) we may consider the following list of rules for it:

\[
\text{eq}(x, x) \rightarrow \text{true} \\
\text{eq}(x, y) \rightarrow \text{false}
\]

Consider two different names \( n \) and \( m \), and assume that \( \text{true} \neq_E \text{false} \). When taking the ordering into account we have \( \text{eq}(n, n) \Downarrow b \iff b =_E \text{true} \). Without the ordering, this is not true since the second rule applies, and thus \( \text{eq}(n, n) \Downarrow \text{false} \).

### 1.1.3 Renaming

A renaming is total applicatio from names to names. Renamings will be noted in the same way as substitutions, implicitly assuming that they behave as the identity where they are not explicitly defined. Their application is also defined analogously. For example, if \( \theta = \{ n \mapsto m, m \mapsto n, p \mapsto n \} \) and \( t = \text{pair}(m, p) \), then \( t\theta = \text{pair}(n, n) \). As shown in that example, a renaming may not be bijective.

**Exercise 3.** Assume that \( s =_E t \) for some \( s, t \in T(N) \). Show that \( s\sigma =_E t\sigma \) for any renaming \( \sigma \).

**Exercise 4.** Consider the variant of computations where reductions rules are ordered. Is it true that \( t \Downarrow u \) implies \( t\sigma \Downarrow u\sigma \) for any renaming \( \sigma \)? If not, propose an extra assumption under which the claim holds.
1.2 Processes

Protocols will be modelled using a process algebra in the style of the applied pi-calculus, which itself elaborates on Milner’s pi-calculus. Although our presentation differs from its specific description, our calculus is compatible with that of Proverif, the main difference being that we do not treat private channels.

We assume a countably infinite set \( C \) of channels, whose elements will be denoted by \( c, d \), etc. Processes are generated from the following grammar:

\[
P, Q ::= 0 \mid (P \mid Q) \mid !P \mid \text{in}(c, x).P \mid \text{out}(c, u).P \mid \text{new} \ n.P \mid \text{let} \ x = t \ \text{in} \ P \ \text{else} \ Q
\]

where \( c \in C \), \( x \in \mathcal{X} \), \( n \in \mathcal{N} \), \( u \in \mathcal{T}(\mathcal{N} \cup \mathcal{X}) \) is a constructor term and \( t \in \mathcal{T}(\mathcal{N} \cup \mathcal{X}) \) is an arbitrary term. Before providing a formal semantics for this language, we describe intuitively what each construct should mean:

- 0 is the process that does nothing.
- \((P \mid Q)\) is the parallel composition of processes \( P \) and \( Q \).
- \(!P\) is the replication of \( P \), which can be thought of as an infinite parallel composition \((P \mid P \mid P \mid \ldots)\).
- \(\text{in}(c, x).P\) is a process that waits for an input on channel \( c \) and then behaves as \( P \) with \( x \) bound to the received message.
- \(\text{out}(c, u).P\) outputs a message \( u \) on \( c \) and then behaves as \( P \).
- \(\text{new} \ n.P\) outputs a message \( u \) on \( c \) and then behaves as \( P \) with \( n \) replaced by \( m \).
- \(\text{let} \ x = y \ \text{in} \ P \ \text{else} \ Q\) attempts to evaluate \( t \) upon success, it binds \( x \) to the resulting message and continues with \( P \); otherwise, it continues with \( Q \).

This syntax is close but not identical to that of Proverif. We refer the reader to the user manual of the tool for the concrete Proverif syntax.

We will consider terms up to associativity and commutativity of parallel composition, and up to the identification of \( P \mid 0 \) and \( P \). This means, for instance, that \((P \mid Q) \mid (R \mid 0)\) and \(Q \mid (P \mid R)\) are the same process, which we would write more simply (and unambiguously) as \( P \mid Q \mid R \).

**Notations.** We will usually omit the null process, writing e.g. \(\text{out}(c, u)\) instead of \(\text{out}(c, u).0\) or \(\text{let} \ x = t \ \text{in} \ P \) instead of \(\text{let} \ x = t \ \text{in} \ P \ \text{else} \ Q\). When \( u, v \in \mathcal{T}(\mathcal{N} \cup \mathcal{X}) \) we write \(\text{if} \ u = v \ \text{then} \ P \ \text{else} \ Q\) for \(\text{let}_\_ = \text{eq}(u, v) \ \text{in} \ P \ \text{else} \ Q\), assuming that \(\text{eq}\) is a destructor defined by the single rule \(\text{eq}(x, x) \rightarrow x\).

**Handling binders.** We write \(\text{fv}(P)\) for the set of free variables of \( P \), i.e. the set of variables that are not bound by an input or a \text{let} construct. Similarly, we write \(\text{fn}(P)\) for the set of free names of \( P \), i.e. the set of names that are not bound by a \text{new} construct. A process \( P \) is closed if \(\text{fv}(P) = \emptyset\) and we will only consider the execution of processes under this condition: this means that when \(\text{out}(c, u)\) is emitted, \(\text{fv}(u) = \emptyset\), hence \( u \) is a message; similarly, when executing \(\text{let} \ x = t \), we have \( t \in \mathcal{T}(\mathcal{N}) \), i.e. it is well-defined to ask whether there exists \( u \) such that \( t \downarrow u \).

The constructs \(\text{in}, \text{let}\) and \text{new} being binders, they induce a notion of \(\alpha\)-renaming. We will implicitly consider terms up to it. As is standard in higher-order rewriting,
we also assume that substitution is capture avoiding (and hence compatible with \(\alpha\)-renaming): this means, for instance, that

\[
\begin{align*}
\text{(new } n. \text{ out}(c, \text{senc}(x, n))).P \{x \mapsto n\} \\
= (\text{new } m. \text{ out}(c, \text{senc}(x, m))).P \{x \mapsto m\} \{x \mapsto n\} \\
= (\text{new } m. \text{ out}(c, \text{senc}(n, m))).P \{x \mapsto m\} \{x \mapsto n\}.
\end{align*}
\]

### 1.2.1 Internal reduction

We first endow processes with an operational semantics that expresses how a closed process may execute.

**Definition 1** (Internal reduction). The binary relation \(P \rightsquigarrow Q\) is given by the following rules:

- \(\text{in}(c, x).P \mid \text{out}(c, u).Q \mid R \rightsquigarrow P\{x \mapsto u\} \mid Q \mid R\)
- \((\text{let } x = t \text{ in } P \text{ else } Q) \mid R \rightsquigarrow P\{x \mapsto u\} \mid R \text{ when } t \Downarrow u\)
- \((\text{let } x = t \text{ in } P \text{ else } Q) \mid R \rightsquigarrow Q \mid R \text{ when } t \Uparrow\)
- \(\text{new } n.P \mid R \rightsquigarrow P\{n \mapsto m\} \mid R \text{ when } m \not\in \text{fn}(\text{new } n.P, R)\)
- \(!P \mid R \rightsquigarrow !P \mid P \mid R\)

**Remark 1.** The following rules are admissible for the syntactic sugar defined above, when \(u \text{ and } v\) are constructor terms – in practice we will use it when \(\Sigma_\alpha = \emptyset\):

- if \(u = v\) then \(P \text{ else } Q) \mid R \rightsquigarrow P \mid R\) when \(u \equiv v\)
- if \(u = v\) then \(P \text{ else } Q) \mid R \rightsquigarrow Q \mid R\) when \(u \not\equiv v\)

The choice of fresh names in the reductions is somewhat arbitrary, as expressed in the following property, where the bijectivity condition is needed even without ordered reduction rules (cf. exercise [1]) due to the presence of else branches.

**Proposition 1.** If \(P \rightsquigarrow Q\) then \(P\sigma \rightsquigarrow Q\sigma\) for any bijective renaming \(\sigma\).

This result will often be used in the case where there is an "undesirable" name \(n \in \text{fn}(P) \setminus \text{fn}(Q)\). Then we can swap \(m\) with any other name \(n \not\in \text{fn}(P) \cup \text{fn}(Q)\), using prop. [1] with the permutation \(\{n \leftrightarrow m\}\), which gives us

\[
P\{n \mapsto m\} = P\{n \leftrightarrow m\} \rightsquigarrow Q\{n \leftrightarrow m\} = Q.
\]

We use this notion of computation — which correctly reflects (some) real-world computations — to give a first formal security definition.

**Definition 2** (Secrecy). A process \(P\) does not ensure the secrecy of a message \(s\) if there exist closed processes \(A\) and \(Q\), a channel \(c\) and a term \(s'\) such that:

- terms in \(A\) belong to \(\mathcal{T}_{\text{pub}}(N)\) and \(\text{fn}(P, s) \cap \text{fn}(A) = \emptyset\),
- \(s \equiv_e s'\) and
- \(P \mid A \rightsquigarrow^* \text{out}(c, s') \mid Q\) and names chosen in this reduction when reducing \text{new} constructs are never taken in the initial "secret set" \(\text{fn}(P, s)\).

Otherwise, we say that \(P\) ensures the secrecy of \(s\).

The condition on free names expresses that the attacker does not know the initial secrets of the protocol. Without it, secrecy would never hold!
Exercise 5. For each of the following processes, indicate when the secrecy of \( n \) is ensured, and exhibit an adversary otherwise:

- \( P_1 = \text{new } k. \text{out}(c, \text{senc}(n, k)). \text{out}(c, k) \)
- \( P_2 = \text{in}(c, x). \text{out}(c, \text{senc}(n, x)) \)
- \( P_3 = \text{out}(c, \text{senc}(n, k)). \text{in}(c, x). \text{if } x = n \text{ then out}(c, k) \)
- \( P_4 = \text{in}(c, x). \text{let } y = \text{adec}(x, k) \text{ in out}(c, k) \text{ else out}(c, \text{aenc}(n, \text{pub}(k))) \)
- \( P_5 = !P_4 \)

Exercise 6. Show that the condition of def. \( 2 \) on the choice of fresh names in the reduction is necessary, but providing an undesirable example that would count as a breach of secrecy without it. Similarly, show the importance of \( \text{fn} \) reduction is necessary, but providing an undesirable example that would count as a breach of secrecy without it. Thus, show that the definition would not adequately model secrecy if \( \text{fn}(P) \) were used instead of \( \text{fn}(P, s) \) in both places.

1.2.2 Labelled transitions

In order to analyze the possible interactions of a process with its environment, it is often more convenient to work with labelled transition semantics, as defined next. In the context of security protocols, it will allow us to characterize secrecy without quantifying over all possible adversaries.

We assume another set \( W \) of special variables called handles and denoted by \( w \). Handles being variables, they are excluded from closed processes. In the labelled transition system, some terms will represent how the adversary may perform a computation involving messages he obtained from the protocol: these terms, called recipes, will belong to \( T_{\text{pub}}(W \cup N) \) and will be denoted by \( R, M, N \).

Definition 3 (Frame). A frame \( \bar{n}, \sigma \) is given by a list of names \( \bar{n} \) and a finite mapping \( \sigma : W \rightarrow T_{\phi}(N) \). Frames are denoted by \( \Phi \) or \( \Psi \). If \( \Phi = \bar{n}, \sigma \) is a frame we write \( \text{bn}(\Phi) \) for \( \bar{n} \); \( \text{dom}(\Phi) \) for \( \text{dom}(\sigma) \); \( \Phi \cup \{ w \mapsto u \} \) for \( \bar{n}, \sigma \cup \{ w \mapsto u \} \); and \( m, \Phi \) for \( \{ m, \bar{n}, \sigma \} \).

Definition 4 (Configuration). A configuration is a pair \( (P, \Phi) \) where \( P \) is a closed process and \( \Phi \) is a frame. Configurations are denoted by \( K \). When \( K \) is a configuration, \( \Phi(K) \) denotes its frame.

We introduce a convenient notation for avoiding heavy freshness conditions on names. Given two objects (terms, processes, frames or sequences of such objects) we write \( x \not\in y \) when no name occurs free in both \( x \) and \( y \), i.e. \( \text{fn}(x) \cap \text{fn}(y) = \emptyset \). For instance, when \( R \) is a recipe, \( R \not\in (P, \Phi) \) means that \( R \in T_{\text{pub}}(W \cup N \setminus \text{fn}(P, \Phi)) \).

Definition 5 (Labelled transitions). The labelled transition relation \( K \rightarrow K' \), given by the rules of fig. \[ ] is a relation between two configurations and an action \( \alpha \) that may be either

- the silent action \( \tau \), or
- the input action \( \text{in}(c, R) \) for some \( c \in C \) and \( R \in T_{\text{pub}}(N \cup W) \), or
- the output action \( \text{out}(c, w) \) for some \( c \in C \) and \( w \in W \).

We define the labelled reflexive transitive closure of \( \rightarrow \) as follows: \( K_0 \rightarrow^* K_n \) when \( \text{tr} = \alpha_1 \ldots \alpha_n \) and \( K_i \rightarrow K_{i+1} \) for all \( i \in [1; n] \).
\((\text{out}(c,u).P \mid Q, \Phi) \xrightarrow{\text{out}(c,w)} (P \mid Q, \Phi \cup \{w \mapsto u\})\) when \(w \notin \text{dom}(\Phi)\)

\((\text{in}(c,x).P \mid Q, \Phi) \xrightarrow{\text{in}(c,R)} (P \{x \mapsto u\} \mid Q, \Phi)\)

when \(R \in \mathcal{T}_{\text{pub}}(\mathcal{N} \cup \text{dom}(\Phi)), R \notin \text{bn}(\Phi)\) and \(R\Phi \downarrow u\)

\((\text{let } x = t \text{ in } P \text{ else } Q \mid R, \Phi) \xrightarrow{} (P \{x \mapsto u\}, \Phi)\) when \(t \downarrow u\)

\((\text{let } x = t \text{ in } P \text{ else } Q \mid R, \Phi) \xrightarrow{} (Q, \Phi)\) when \(t \not\in\)

\((\text{new } n.P \mid Q, \Phi) \xrightarrow{} (P \{n \mapsto m\} \mid Q, m, \Phi)\) if \(m \notin (\text{new } n.P, Q, \Phi, \text{bn}(\Phi))\)

\((!P \mid Q, \Phi) \xrightarrow{} (!P \mid P \mid Q, \Phi)\)

Figure 1: Labelled transitions between configurations

The main novelty here is that, when \(K = (P, \Phi)\) performs a labelled transition, communication is not taking place between sub-processes of \(P\). Instead, the transition represents a possible interaction with an hypothetical environment (or attacker, in our context) whose knowledge is represented by \(\Phi\). This idea is pushed to the extreme here, and sub-processes of \(P\) are not even allowed to communicate. The intuitive justification is that, since the attacker can eavesdrop and inject messages, he can in particular mediate internal communications, and we might as well assume that he mediates them all. Formally, adding the internal communication rule would not change the next result.

**Proposition 2.** If \(K \xrightarrow{} K'\) and \(\sigma\) is a bijection on names, then \(K\sigma \xrightarrow{\sigma} K'\sigma\).

We now formulate the analogue of secrecy in the framework of frames and labelled transitions, before establishing that it captures the same idea.

**Definition 6** (\(\Phi \vdash u\)). A frame \(\Phi\) allows to deduce a message \(s\), written \(\Phi \vdash s\), if there exists a recipe \(R\) such that \(R \notin \text{bn}(\Phi)\) and \(R\Phi \downarrow s\).

**Proposition 3** (Secrecy). A process \(P\) does not ensure the secrecy of \(s\) iff there exist \(tr, P', \Phi'\) such that \((P, \text{ln}(P,s),\emptyset) \xrightarrow{} (P', \Phi')\) and \(\Phi' \vdash s\).

**Proof.** We first prove that, if there exists an execution \((P, \Phi) \xrightarrow{} (P', \Phi')\) and a recipe \(R \notin \text{bn}(\Phi)\) such that \(R\Phi' \downarrow s\), then there exists a process \(A\) containing terms in \(\mathcal{T}_{\text{pub}}(\text{dom}(\Phi) \cup \mathcal{N} \setminus \text{bn}(\Phi))\) such that \(P \mid A\Phi \rightarrow^* \text{out}(c,s)\) \(\ldots\) with a reduction that does not pick fresh names in \(\text{bn}(\Phi)\).

We proceed by induction on \(tr\), essentially translating \((tr, R)\) into an attacker \(A\). If \(tr = \varepsilon\) we have \(R \notin \text{bn}(\Phi)\) and \(R\Phi \downarrow s\). We conclude with \(A := \text{let } x = R\text{ in }\text{out}(c, x)\) since \(A\Phi \rightarrow \text{out}(c, s)\).

Assume now that \(tr = \alpha.tr'\). We have \((P, \Phi) \xrightarrow{} (P_1, \Phi_1) \xrightarrow{} (P', \Phi')\) and we proceed by case analysis on the first transition.

- If \(\alpha = \text{out}(c, w)\), \(P\) is of the form \(\text{out}(c,v).Q \mid R, P_1 = Q \mid R\) and \(\Phi_1 = \Phi \cup \{w \mapsto v\}\) for some \(w \notin \text{dom}(\Phi)\). By induction hypothesis on \(tr'\) we have an adversary \(A_1\) against \(P_1\) such that \(A_1 \notin \text{bn}(\Phi_1)\). We conclude with \(A := \text{in}(c,w).A_1\): we check that \(A \notin \text{bn}(\Phi)\) = \(\text{bn}(\Phi_1)\); that terms of \(A\) are in \(\mathcal{T}_{\text{pub}}(\mathcal{N} \cup \text{dom}(\Phi))\), because \(\text{dom}(\Phi) = \text{dom}(\Phi_1) \setminus \{w\}\); and that the expected reduction is possible, because the input of \(A\) and the output of \(P\), both on channel \(c\), can be used in a communication rule so that \(P \mid A\Phi \rightarrow P_1 \mid A_1\Phi \{w \mapsto v\} \rightarrow P_1 \mid A_1\Phi\).
• If \( \alpha = \text{in}(c, R) \) we have \( R \not\ni \text{bn}(\Phi), R\Phi \Downarrow u \) and
\[
(P, \Phi) = (\text{in}(c, x)Q \mid R, \Phi) \Rightarrow (Q(x \mapsto u) \mid R, \Phi) = (P_1, \Phi_1).
\]
We obtain by induction hypothesis an adversary \( A_1 \) against \( P_1 \). The attacker
\( A \coloneqq \text{let } x = R \text{ in out}(c, x).A_1 \) (for some \( x \not\in \text{fv}(A_1) \)) allows us to con-
due. We easily check that it executes well, and that it contains the same free
variables as \( A_1 \). It also satisfies \( A \not\ni \text{bn}(\Phi) \) since \( R \not\ni \text{bn}(\Phi) \).

• If the first transition is a name creation, \( P \) is of the form \( \text{new } n.Q \mid R \) and
\( P_1 = Q[n \mapsto m] \mid R \) for some \( m \not\in (P, \Phi, \text{bn}(\Phi)) \). By induction hypothesis
we obtain an adversary \( A_1 \) against \( P_1 \), satisfying \( A_1 \not\ni \text{bn}(\Phi_1) \) and hence
\( A_1 \not\ni \text{bn}(\Phi) \) since \( \Phi_1 = m.\Phi \). Since \( m \not\in \text{bn}(A_1) \), we can reduce \( P \mid A_1.\Phi \leadsto P_1 \mid A_1.\Phi_1 \), which allows us to conclude with
\( A \coloneqq A_1 \).

• The other cases, i.e. \( \text{let } \) evaluations and replication, are similar.

For the other direction, we need to find an inductive characterization of secrecy
(def. 2). The problem is the notion of adversary: the condition \( A \not\ni P \) is not preserved through the reductions of \( P \mid A \). Frames get us closer to a solution: in general, when
considering a process \( P \) and a current frame \( \Phi \), it would seem reasonable to consider
adversaries of the form \( A\Phi \) where \( A \not\ni P \) and all terms in \( A \) belong to \( T_{\text{pub}}(N \cup W) \).
But this form of adversaries is not stable by reduction of \( \text{let } \) constructs, so we need
to introduce a slightly more complex notion.

In the proof below, we call adversary a closed process in which some (sub)terms
may be decorated by a recipe, which is noted \( u^R \). We say that an adversary \( A \) is a
\( \Phi \)-adversary when \( R\Phi \Downarrow u \) for all decorated subterms \( u^R \) occurring in \( A \). Given a
term \( t \) with possibly decorated subterms, \( R(t) \) is obtained from \( t \) by replacing any
\( u^R \) by \( R \). By extension, if \( A \) is an adversary, \( R(A) \) is obtained by replacing each
\( u^R \) by \( R \) — note that the resulting object might not be a well-formed process, e.g.
since destructors may occur in output terms. For adversaries, freshness conditions
are always wrt. \( R(A) \), i.e. \( A \not\ni X \) means \( R(A) \not\ni X \). Intuitively, we do not consider
the computed terms but only the recipes that have been used to obtain them. In
the same spirit, we require that any \( \Phi \)-adversary \( A \) is such that \( R(A) \) only contains
terms in \( T_{\text{pub}}(N \cup \text{dom}(\Phi)) \).

As an exercise, one can check that the existence of an adversary in the sense of
def. 2 is equivalent to the existence of an \( \emptyset \)-adversary. Thus, it suffices to establish
that there exists a trace \( (P, \Phi) \Rightarrow^* (P', \Phi') \) such that \( \Phi' \vdash s \) whenever there exists a
\( \Phi \)-adversary \( A \) such that \( A \not\ni \text{bn}(\Phi) \) and \( P \mid A \leadsto^* \text{out}(\_ s') \mid \ldots \) for some \( s' =_{E} s \),
using a reduction that does not pick fresh names in \( \text{bn}(\Phi) \).

We proceed by induction on (the length of) the execution of \( P \mid A \).

• If the reduction sequence is empty then \( \text{out}(c, s') \) is an immediate parallel
sub-process of \( P \mid A \): if it belongs to \( P \) we conclude with \( tr = \text{out}(c, w) \) for
some \( w \) since \( \Phi \cup \{ w \mapsto s' \} \vdash s \) by taking \( R = w; \) if it belongs to \( A \)
we have \( R(s') \Downarrow s \), hence \( \Phi \vdash s \), so we can conclude with \( tr = \epsilon \) because
\( R(s') \not\ni \text{bn}(\Phi) \) inherited from \( A \not\ni \text{bn}(\Phi) \).

• If the first reduction step is internal to \( A \), i.e. \( P \mid A \leadsto P' \mid A' \) with \( A \leadsto A' \),
we conclude by induction hypothesis on the rest of the reduction involving \( A' \),
keeping \( tr \) unchanged. If the reduction is a name creation, we have assumed
that the chosen name satisfies \( m \not\in \text{bn}(\Phi) \), hence \( A' \not\ni \text{bn}(\Phi) \). If the reduction
is the computation of some \( \text{let } x = t \), we also need to check that \( A' \) is still a
\( \Phi \)-adversary: after computing \( t \Downarrow u \), we actually replace \( x \) by \( u^{R(t)} \) to justify
the subterm \( u - \) indeed, we have \( R(t)\Phi \Downarrow u \).

\( \triangleright \) In particular, a \( \Phi \)-adversary
where all decorated terms are
of the form \( \Phi(w)^w \) is simply a
process of the form \( A\Phi \).

\( \triangleright \) If \( t \) occurs in a \( \Phi \)-adversary
and \( t \Downarrow u \), then \( R(t)\Phi \Downarrow u \).
• Assume now that the first step is a communication occurring inside \( P \), exchanging \( u \) on channel \( c \). We have

\[
(P, \Phi) \xrightarrow{\text{out}(c,u) \cdot \text{in}(c,w)} (P', \Phi \cup \{ w \mapsto u \})
\]

which allows us to conclude by induction hypothesis with \( A \), which is also a \((\Phi \cup \{ w \mapsto u \})\)-adversary against \( P' \).

• For other reductions internal to \( P \) we conclude by induction hypothesis with \( \tau = \tau' \). If \( P \) creates a name \( m \), we need \( A \) to be a \((m,\Phi)\)-adversary satisfying \( A \not\in \text{bn}(m,\Phi) \): the former is immediate, the latter comes from the side condition on the reduction of \( P | A \).

• Assume that, in the first reduction step, a message \( u \) is sent by \( A \) and received by \( P \) on some channel \( c \). Since \( A \not\in \text{bn}(\Phi) \) we have \( R(u) \not\in \text{bn}(\Phi) \) and thus

\[
(P, \Phi) \xrightarrow{\text{in}(c,R(u))} (P' \{ x \mapsto u \}, \Phi).
\]

We conclude by induction hypothesis using \( P' \{ x \mapsto u \} \) and \( A' \): one easily checks that \( A' \) is still a \( \Phi \)-adversary, and \( A' \not\in \text{bn}(\Phi) \).

• Consider finally the case where \( P \) sends some message \( u \) to \( A \). Then

\[
(P, \Phi) \xrightarrow{\text{out}(c,u)} (P', \Phi \cup \{ w \mapsto u \})
\]

and we conclude by induction hypothesis using \( P' \) and \( A' \{ x \mapsto u \} \).

\( \square \)

**Exercise 7.** For each process \( P \) of exercise 3 exhibit (if it exists) an execution \((P, \emptyset) \xrightarrow{tr} (Q, \Phi)\) and a recipe \( R \uparrow \text{bn}(\Phi) \) such that \( R\Phi \Downarrow n \).
2 Verifying secrecy for bounded executions

In this section we will present an important approach to verify secrecy, based on (for-
ward) symbolic execution and constraint solving. This approach is only applicable
when process executions are bounded, and has been superseded by other methods
for secrecy and other reachability properties, but it remains interesting for its preci-
sion, for the applicability of its main concepts for checking equivalence, and for its
general interest in program verification.

The rest of the section follows the main steps of the method. First, the process
under study is executed symbolically (section 2.2) to obtain a finite number of reach-
able symbolic states, each such state coming with a set of deducibility constraints.
Then, secrecy is checked in each symbolic state by means of a constraint solving
procedure (section 2.3). Before covering these symbolic problems, we will need to
consider the simpler deduction problem (section 2.1).

In the rest of this section we will assume that the signature contains symbols for
representing pairing and asymmetric encryption, both modelled using destructors.
In other words we have
\[ \Sigma_c = \{ \text{pair}, \text{aenc} \} \]
and
\[ \Sigma_d = \{ \text{proj}_1, \text{proj}_2, \text{adec} \} \]
no equation, and the following reduction rules:

\[ \text{proj}_i(\text{pair}(x_1, x_2)) \rightarrow x_i \quad \text{adec}(\text{aenc}(x, \text{pub}(y)), y) \rightarrow x \]

2.1 Deciding deduction

In the concrete semantics of fig. 1 we have been using recipes to witness how an
attacker might derive a given message. However, when checking reachability prop-
erties such as secrecy, we only care about which terms are derivable and not how
they are derived. Another way to say this is that, when checking for secrecy for
a given configuration, we only need to consider a single recipe for a given input
message. It is important, because a given message always admits infinitely many
recipes, and we would like to be able to restrict to a finite set of small recipes.

One way to forget recipes is to consider an intruder deduction system, such as
the one of fig. 2 for the primitives considered here. The first rule is called the axiom
rule, then there is, for each constructor \( f \in \Sigma_c = \{ \text{aenc, pair} \} \), a composition
rule that allow to deduce new messages with toplevel symbol \( f \) and a decomposition
rule that allow to deduce subterms of messages with toplevel symbol \( f \).

\[ \Phi(w) = u \quad \Phi \vdash w : u \quad n \in \mathcal{N} \setminus \text{bn}(\Phi) \]
\[ \Phi \vdash n : n \]
\[ \Phi \vdash R : u \quad \Phi \vdash \text{pub}(R) : \text{pub}(u) \]
\[ \Phi \vdash \text{pair}(R_1, R_2) : \text{pair}(u_1, u_2) \]
\[ \Phi \vdash R_1 : u_1 \quad \Phi \vdash R_2 : u_2 \]
\[ \Phi \vdash \text{aenc}(R_1, R_2) : \text{aenc}(u, v) \]
\[ \Phi \vdash R_1 : \text{aenc}(u, \text{pub}(v)) \quad \Phi \vdash R_2 : v \]
\[ \Phi \vdash \text{adec}(R_1, R_2) : u \]

Figure 2: Decorated deduction rules.

Proposition 4. Let \( \Phi \) be a frame. We have \( \Phi \vdash u \) (in the sense of def. 2) iff \( \Phi \vdash R : u \)
is derivable (in the sense of fig. 3).

When one is concerned only with the deducibility of \( u \) from \( \Phi \), i.e. when the
specific recipe is irrelevant, the derivation system of fig. 2 can be simplified to omit
recipes. Instead of deriving judgments of the form \( \Phi \vdash R : u \), judgments are simply
of the form \( \Phi \vdash u \) and each rule is then adapted in the straightforward way. For example, the first two rules becomes:

\[
\begin{array}{c}
T \vdash u \\
\Phi \vdash u \\
N \setminus \text{bn} \left( \Phi \right) \vdash n \\
\Phi \vdash n
\end{array}
\]

Also observe that \( \Phi \) never changes during the course of a derivation, hence it can sometimes be omitted if it is clear from the context.

**Proposition 5.** The problem of deciding \( \Phi \vdash u \) given \( \Phi \) and \( u \) is in PTIME.

**Proof sketch.** If there is a derivation of \( \Phi \vdash u \), then there is a derivation in which no two points of a branch derive the same message. In derivations without repetitions, a composition rule cannot be used to deduce the left premise of a judgment, as in the following example:

\[
\begin{array}{c}
T \vdash v_1 \\
T \vdash v_2 \\
\text{pair}(v_1, v_2) \vdash \text{pair}(v_1, v_2)
\end{array}
\]

To conclude, it suffices to observe that, in derivations without repetitions, all statements \( \Phi \vdash v \) are such that \( v \) is either a subterm of \( v \) (the conclusion) or of \( \Phi \). \( \Box \)

### 2.2 Symbolic execution

Our goal here is to design a symbolic LTS that allows to finitely (and precisely) describe the possible transitions of the LTS of fig. 1. Compared to that LTS, to which we will refer as the concrete LTS, the symbolic LTS will allow free variables in its configurations. To control the possible instantiations of these free variables, symbolic configurations will also feature constraints.

In order to symbolically describe (un)successful computations, we define in figs. 3 and 4 the relations \( t \Downarrow^\varphi x \) and \( t \Downarrow^\varphi \perp \) where \( t \in T(N \cup X) \), \( x \in X \) and \( \varphi \) is a conjunction of equalities between terms of \( T_e(N \cup X) \).

**Example 6.** We can derive \( \text{aenc} \left( \text{proj}_1(x, y) \right) \Downarrow^\varphi z \) with

\[
\varphi := x = x \land x = \text{pair}(x_1, x_2) \land x_1 = x_1 \land y = y \land x_1 = \text{aenc}(z, \text{pub}(y)).
\]

This constraint is equivalent to \( x = \text{pair}(\text{aenc}(z, \text{pub}(y)), x_2) \).

**Example 7.** We can derive (up to constraint simplification) \( \text{proj}_1 \left( \text{pair}(x_1, \text{proj}_1(x_2)) \right) \Downarrow^\varphi y \) with

\[
\varphi := x_2 = \text{pair}(x_2', x_2'') \land y' = x_2' \land y'' = \text{pair}(x_1, y') \land y = x_1.
\]

Note that the constraint imposes that \( x_2 \) is a pair, even though it is ignored in the result.

In order to express the associated soundness and completeness statements, we define \( \text{Sol}(\varphi) \) as the set of all \( \theta :fv(\varphi) \rightarrow T_e(N) \) such that \( \varphi \theta \) is a conjunction of identities; we also write \( \theta \subseteq \theta' \) when \( \text{dom}(\theta) \subseteq \text{dom}(\theta') \) and \( \theta'|_{\text{dom}(\theta)} = \theta \).

**Proposition 6.** Let \( t \in T(N \cup X) \) and \( x \in X \).

- For every \( t \Downarrow^\varphi x \) and \( \theta \in \text{Sol}(\varphi) \), \( t \theta \Downarrow \theta(x) \).

- For every \( \theta :fv(t) \rightarrow T_e(N) \) and \( u \) such that \( t \theta \Downarrow u \), there exists \( \varphi \) such that \( t \Downarrow^\varphi x \), \( \theta \subseteq \theta' \in \text{Sol}(\varphi) \) and \( u = \theta'(x) \).

**Proposition 7.** Let \( t \in T(N \cup X) \).

- For every \( t \Downarrow^\varphi \perp \) and \( \theta \in \text{Sol}(\varphi) \), \( t \theta \not\Downarrow \).

12
\[
\frac{y \Downarrow x = y}{t_1 \Downarrow \varphi_1 \ x_1}, \frac{t_2 \Downarrow \varphi_2 \ x_2}{f(t_1, t_2) \Downarrow \varphi_1 \land \varphi_2 \land x = f(x_1, x_2)} \quad f \in \{\text{pair, aenc}\}
\]

\[
\frac{\text{proj}_i(t) \Downarrow \varphi \land \text{pair}(x_1, x_2) \land x = x_1}{t_1 \Downarrow \varphi_1 \ x_1}, \frac{t_2 \Downarrow \varphi_2 \ x_2}{\text{adec}(t_1, t_2) \Downarrow \varphi_1 \land \varphi_2 \land x_1 = \text{aenc}(x, \text{pub}(x_2)) \ x}
\]

Figure 3: Symbolic computation success rules

\[
\frac{t_1 \Downarrow \varphi \perp}{f(t_1, t_2) \Downarrow \varphi \perp} \quad f \in \{\text{pair, aenc}\}
\]

\[
\frac{\text{proj}_i(t) \Downarrow \varphi \perp}{f(t_1, t_2) \Downarrow \varphi \perp} \quad f \in \{\text{pair, aenc}\}
\]

\[
\frac{\text{proj}_i(t) \Downarrow \varphi \perp}{\text{adec}(t, k) \Downarrow \varphi \perp}
\]

\[
\frac{\text{proj}_i(t) \Downarrow \varphi \perp}{\text{adec}(t, k) \Downarrow \varphi \perp}
\]

Figure 4: Symbolic computation failure rules

- For every \( \theta : \text{fv}(t) \rightarrow \mathcal{T}_c(\mathcal{N}) \) such that \( t \not\Downarrow \varphi \), there exists \( \varphi \) such that \( t \Downarrow \varphi \ x \) and \( \theta \subseteq \theta' \in \text{Sol}(\varphi) \).

Note that our rules do not impose any freshness condition on the result variables. This is not necessary for soundness. However, choosing fresh result variables (e.g. \( x_1 \) and \( x_2 \) in the rule for \( f(t_1, t_2) \)) is interesting in practice as it is sufficient to obtain completeness. This can be understood by considering an example: \( \text{pair}(x, y) \Downarrow \varphi \ x \) can only be derived with a \( \varphi \) that admits no solution, while \( \text{pair}(x, y) \Downarrow x = x' \land y = y' \land \text{pair}(x', y') \) accounts for all solutions.

Even if the choice of variables is made finite by taking a single fresh variable each time, there remains an infinite number of choices for the name \( n \) in the second rules for the failing deconstructors. This could be avoided in practice by choosing one specific name, or using a special name(t) predicate, but we do not detail such a treatment.

We are now ready to define the symbolic semantics. It is given in fig. 5 in the form of a labelled transition system over symbolic configurations that feature a constraint system.

**Definition 7.** A deducibility constraint system is a finite set of deducibility constraints of the form \( T \vdash \perp \ u \) where \( T \) is a set of constructor terms and \( u \) is a constructor term. A system is viewed as a conjunction of constraints, thus the empty system is written \( \perp \) and the conjunction symbol is used to denote the union of constraint systems. We assume two conditions on a deducibility constraint system \( T_1 \vdash \perp \ u_1 \land \ldots \land T_n \vdash \perp \ u_n \):

- Monotonicity: \( T_1 \subseteq \ldots \subseteq T_n \).
- Origination: \( \text{fv}(T_{i+1}) \subseteq \text{fv}(u_1, \ldots, u_i) \) for all \( 1 \leq i < n \).

**Definition 8.** Let \( C \) be a constraint system. We say that \( \theta \in \text{Sol}(C) \) when \( \theta : \text{fv}(C) \rightarrow \mathcal{T}_c(\mathcal{N}) \) and, for each constraint \( T \vdash \perp \ u \) of \( C \), \( T \theta \vdash \ u \theta \).

**Definition 9.** A symbolic configuration is a tuple \( \langle P, \Phi, C \rangle \) where \( P \) is a process with free variables, \( \Phi = \vec{n}.\sigma \) is a frame whose messages may contain free variables, and \( C \) is a constraint system.
\[ K = (\text{in}(c,x).P \mid Q, \Phi, C) \xrightarrow{\text{in}(c,x)} (P \mid Q, \Phi, C \land \Phi \vdash x) \quad x \not\in \text{fv}(K) \]
\[ (\text{out}(c,u).P \mid Q, \Phi, C) \xrightarrow{\text{out}(c,u)} (P \mid Q, \Phi \cup \{w \mapsto u\}, C) \quad w \not\in \text{dom}(\Phi) \]
\[ \text{let } x = t \text{ in } P \text{ else } Q \mid R, \Phi, C \Rightarrow (P \mid R, \Phi, C) \theta \quad t \Downarrow^{\phi} x, \quad \theta \in \text{Sol}(\phi) \]
\[ \text{let } x = t \text{ in } P \text{ else } Q \mid R, \Phi, C \Rightarrow (Q \mid R, \Phi, C) \theta \quad t \Downarrow^{\phi} \bot, \quad \theta \in \text{Sol}(\phi) \]
\[ K = (\text{new} n.P \mid Q, \Phi, C) \Rightarrow (P \mid Q, n, \Phi, C) \quad n \not\in K \]
\[ ([P \mid Q, \Phi, C] \Rightarrow (P \mid ![P \mid Q, \Phi, C])) \]

Figure 5: Symbolic labelled transitions.

When \( K \) is a symbolic configuration \((P, \Phi, C)\), we simply write \( \text{Sol}(K) \) for \( \text{Sol}(C) \). Given a concrete configuration \( K = (P, \Phi) \), we define \( [K] \) as the symbolic configuration \((P, \Phi, \top)\). Conversely, if \( K' = (P, \Phi, C) \) is a symbolic configuration and \( \theta \in \text{Sol}(K') \), we define \([K']\theta \) as the concrete configuration \((P\theta, \Phi\theta)\).

**Proposition 8** (Soundness and completeness). Let \( K_1 \) be a (concrete) configuration.

- If \( [K_1] \xrightarrow{t} K_2' \) and \( \theta \in \text{Sol}(K_2') \), then \( K_1 \xrightarrow{t} [K_2']\theta \) for some \( t \).
- If \( K_1 \Rightarrow K_2 \) then there exists \( K_2' \) and \( \theta \in \text{Sol}(K_2') \) such that \( [K_1] \xrightarrow{t} K_2' \) and \( K_2 = [K_2']\theta \).

Moreover, \( t \) and \( t' \) only differ in input actions: when \( x \) occurs in an input of \( t' \), a recipe that allows to derive \( \theta(x) \) occurs in the corresponding position of \( t' \).

### 2.3 Solving deducibility constraint systems

We refer the reader to chapter 3 of the old lecture notes for this part. There are a few superficial differences:

- The old lecture notes consider symmetric encryption, but this can simply be ignored. (I avoided it because I prefer to consider it as being given via constructors and equations, and I did not want to develop the symbolic semantics with the induced extra complexity.)

- The old lecture notes feature a deduction system with a name rule. This is because it considers a setting where the attacker cannot generate names at all, and variables can be used instead when needed: recall that the deduction system is used, in the end, to deduce terms with free variables from terms with free variables.

- The old lecture notes feature a deduction system with judgments of the form \( u \) rather than \( \Phi \vdash u \). This is mostly a simple change of viewpoint: instead of deriving \( \Phi \vdash u \) one seeks to derive \( u \) with a derivation featuring open (unjustified) leaves labelled with terms in \( \text{img}(\Phi) \).

- Instead of considering a reduction \( \text{adec}(\text{aenc}(x, pub(y)), y) \rightarrow x \), the lectures notes work with \( \text{adec}(\text{aenc}(x, y), \text{sk}(y)) \rightarrow x \) where \( \text{sk} \) is a private function symbol. As a result of this difference, we need one more rule (cf. my slides) in the constraint solving procedure. We also need a slightly modified second lemma: see my slides for the modified statement, and try to update the proof as an exercise.

\hspace{1cm} \text{http://www.lsv.fr/~baelde/secu/poly.pdf}
3 Verifying secrecy for unbounded executions

4 Equivalences

5 Verifying equivalences

6 Advanced topics