We present in chapter 1 a symbolic model of security protocols, and illustrate how it can be used, e.g. to model secrecy. Secrecy verification is developed in more detail in chapters 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 that Proverif uses in the unbounded case, using a Horn-clause abstraction of protocols. We go back to the semantics of our processes in chapter 4 to define and study various behavioral equivalences, and show how they are useful to model more advanced security properties: strong secrecy, anonymity, unlinkability, etc. We will only briefly describe how equivalences may be verified, in particular in Proverif. Finally, chapter 5 introduces a much more recent approach which allows to use symbolic techniques to obtain security proofs in the computational model.
Chapter 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) $\pi$-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 $\text{ar}_\Sigma : \Sigma \to \mathbb{N}$. 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 $\text{senc}$ and $\text{sdec}$, of arity 2, represent symmetric encryption and decryption. Pairing is modeled using $\text{pair}$ of arity 2 and projection functions $\text{proj}_1$ and $\text{proj}_2$, both of arity 1. Finally, the symbol $\text{ok}$ is of arity 0, i.e. it is a constant.

With this signature we have $\text{pair(\text{ok, ok})} \in T(\emptyset)$, $\text{senc(\text{pair(ok, ok)}, k)} \in T(N)$ and $\text{sdec(\text{ok, x})} \in T(X)$. However, $\text{senc(\text{ok})}$ and $\text{ok(\text{ok})}$ are not terms.

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

Given a term $t$, we define $\text{fv}(t)$ as the set of variables that occur in $t$. Similarly, $\text{fn}(t)$ 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 $\text{dom}(\cdot)$. The application of a substitution $\theta$ to a term $t$ is defined as usual and noted $t\theta$. In particular, when $\text{dom}(\theta) \cap \text{fv}(t) = \emptyset$, $t\theta = t$.

We will introduce two mechanisms for giving a meaning to function symbols. First, we will introduce equations to model when two terms should be considered equal, i.e. when they represent computations that yield the same result. For example, we may equate $\text{proj}_1(\text{pair}(s, t))$ and $s$. Second, we provide our term algebra

Names will be used to represent the secrets of honest participants: nonces, keys, identities, etc. The attacker will not know them a priori. Function symbols will represent specific terms, terms constructions or computations over terms. Variables, as usual, will be used as placeholders for unknown terms.
with a means to describe computations that may fail. For example, we may have that \texttt{sdec(senc}(s, k), k) reduces to \texttt{s} but \texttt{sdec(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 constructor and destructor symbols, i.e. \( \Sigma = \Sigma_c \uplus \Sigma_d \). 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 \( E \subseteq T_c(\mathcal{X})^2 \); we will use an infix notation for it, writing \( s \equiv t \) rather than \( (s, t) \in E \). We then define the binary relation \( =_E \) over \( T_c(N \cup \mathcal{X}) \) as the least equivalence relation that contains \( E \) and is closed under substitution and context closure. In other words, we impose that:

- for all \( s \equiv t \), we have \( s \equiv E t \);
- for all \( s \equiv E t \) and for any substitution \( \theta : \mathcal{X} \rightarrow T_c(N \cup \mathcal{X}) \), we have \( s\theta \equiv E t\theta \);
- 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 example\(^7\) and assuming that \( \Sigma = \Sigma_c \), consider \( E \) made of three equations:

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

We then have \( \text{proj}_j(\text{sdec(senc(pair(ok, n), k))) =_E ok but ok \neq E pair(ok, n)} \).

**Exercise 1.** Let \( u \) and \( v \) be terms such that \( u \equiv E v \), and let \( x \) be a variable. Show that \( t\{x \mapsto u\} \equiv E t\{x \mapsto v\} \) for any term \( t \). Conclude that the substitution principle holds: for any terms \( s, t, u, v \) and variable \( x \), \( s \equiv E t \) and \( u \equiv E v \) imply \( s\{x \mapsto u\} \equiv E t\{x \mapsto v\} \).

### 1.1.2 Rewrite rules

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

- for all \( n \in N \), \( n \Downarrow n \);
- for all \( f \in \Sigma_c \) with \( \text{ar}(f) = n \),
  for all \( u_1, \ldots, u_n \) such that \( u_i \Downarrow u_i \) for all \( i \in [1; n] \),
  \( f(u_1, \ldots, u_n) \Downarrow f(u_1, \ldots, u_n) \);
- for all \( f \in \Sigma_d \) with reduction rule \( f(u_1, \ldots, u_n) \rightarrow u \),
  for all \( u_1, \ldots, u_n \) and \( \theta : \mathcal{X} \rightarrow T_c(\mathcal{N}) \) such that \( u_i \Downarrow u_i\theta \) for each \( i \in [1; n] \),
  \( f(u_1, \ldots, u_n) \Downarrow u\theta \);
- for all \( t, u \) and \( v \) such that \( t \Downarrow u \) and \( u \equiv E v \), \( t \Downarrow v \).

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

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

\( \star \) For practical applications, equations are often oriented into rewrite rules, and good properties (e.g. confluence, termination) are required to obtain e.g. computable equality and unification modulo \( E \).

\( \star \) 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 \( \equiv_E \)) or computable.
Example 3. Consider again $\Sigma$ from example 1 but assuming now that $\Sigma_d = \{\text{proj}_1, \text{proj}_2\}$.
Assume that $E$ contains only the equation $\text{sdec}(\text{senc}(x, y), y) \in x$, 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
\[
\text{proj}_1(\text{sdec}(\text{senc}(\text{pair}(\text{ok}, n), k), k)) \not\rightarrow \text{ok}.
\]
Observe that the fourth item of the definition of $\not\rightarrow$ is crucial to obtain this computation, as it is needed to have $\text{sdec}(\text{senc}(\text{pair}(\text{ok}, n), k), k) \not\rightarrow \text{pair}(\text{ok}, n)$. We also have $\text{pair}(\text{ok}, n) \not\rightarrow \text{ok}$: in fact, $\text{pair}(\text{ok}, n) \not\rightarrow u$ iff $u =_E \text{pair}(\text{ok}, n)$. Finally, there are terms that cannot be computed, e.g. $\text{proj}_1(\text{ok}) \not\rightarrow$ and $\text{proj}_1(\langle \text{ok}, \text{proj}_1(\text{ok}) \rangle) \not\rightarrow$.

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 $t \not\rightarrow w$: 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 $pk$, and the following reduction rule:
\[
\text{adec}(\text{aenc}(x, pk(y)), y) \rightarrow x.
\]

Example 5. Equality tests can be obtained by taking a destructor $\text{eq}$ equipped with a single reduction: $\text{eq}(x, x) \rightarrow x$. Indeed, one has $\text{eq}(s, t) \not\rightarrow u$ iff $s \not\rightarrow u$ and $t \not\rightarrow u$.

Example 6. In several settings, it is required that destructors are deterministic up-to the equational theory, so that $t \not\rightarrow u$ and $t \not\rightarrow v$ iff $u =_E v$. This is not forced in our definition. For instance, we can define a typical non-deterministic binary destructor $\text{choose}$ by the following two reductions:
\[
\text{choose}(x, y) \rightarrow x
\]
\[
\text{choose}(x, y) \rightarrow y
\]

Example 7. 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, we may equip a destructor $\text{eqb}$ with the following list of rules:
\[
\text{eqb}(x, x) \rightarrow \text{true}
\]
\[
\text{eqb}(x, y) \rightarrow \text{false}
\]
Consider two different names $n$ and $m$, and assume that $\text{true} \not\rightarrow _E \text{false}$. When taking the ordering into account we have $\text{eqb}(n, n) \not\rightarrow b$ iff $b =_E \text{true}$. Without the ordering, this is not true since the second rule applies, and thus $\text{eqb}(n, n) \not\rightarrow \text{false}$.

1.1.3 Renaming

A renaming is a total application 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 2. Assume that \( s =_E t \) for some \( s, t \in T(N) \). Show that \( s\sigma =_E t\sigma \) for any renaming \( \sigma \).

Exercise 3. 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:

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

where \( c \in C, x \in X, n \in N, u \in T(N \cup X) \) is a constructor term and \( t \in T(N \cup X) \) 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.
- \( (P | Q) \) 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 \((P | P | P | \ldots)\).
- \( \text{in}(c, x).P \) 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 \( P \).
- \( \text{new} \ n.P \) creates a new (previously unused) name \( m \) and then behaves as \( P \) with \( n \) replaced by \( m \).
- \( \text{let} \ x = t \ \text{in} \ P \ \text{else} \ Q \) 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 \( P \ | \ 0 \) and \( P \). This means, for instance, that \((P | Q) \ | \ (R \ | \ 0) \) and \( Q \ | \ (P \ | \ R) \) are the same process, which we would write more simply (and unambiguously) as \( P \ | \ Q \ | \ R \).

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 \( u, v \in T(N \cup X) \) 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 eq is defined as in example 5.
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 \textbf{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 \textbf{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 \( t \in T(N) \), i.e. it is well-defined to ask whether there exists \( u \) such that \( t \downarrow u \).

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

\[
\left( \text{new} \ m. \ \text{out}(c, \text{senc}(x, m)).P \right) \{ x \mapsto n \} = \left( \text{new} \ m. \ \text{out}(c, \text{senc}(x, m)).P \{ n \mapsto m \} \right) \{ x \mapsto n \} = \left( \text{new} \ m. \ \text{out}(c, \text{senc}(n, m)).P \{ n \mapsto m \} \{ x \mapsto n \} \right).
\]

### 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:

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

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

\[
\begin{align*}
\text{if } u = v \text{ then } P \mid \text{else } Q \mid R \rightsquigarrow P \mid R & \quad \text{when } u =_E v \\
\text{if } u = v \text{ then } P \mid \text{else } Q \mid R \rightsquigarrow Q \mid R & \quad \text{when } u \neq _E v 
\end{align*}
\]

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 \footnote{Private channels allow another simple encoding of this common operator. Even without non-deterministic destructors and private channels, more complex encodings of non-deterministic choice of processes are possible.} due to the presence of \textbf{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 \( n \) with any other name \( m \notin \text{fn}(P) \cup \text{fn}(Q) \), using proposition \footnote{Private channels allow another simple encoding of this common operator. Even without non-deterministic destructors and private channels, more complex encodings of non-deterministic choice of processes are possible.} with the permutation \( \{ n \mapsto m \} \), which gives us:

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

**Example 8.** Using the non-deterministic destructor of example \footnote{Private channels allow another simple encoding of this common operator. Even without non-deterministic destructors and private channels, more complex encodings of non-deterministic choice of processes are possible.} we can encode non-deterministic choice in processes:

\[
P + Q \overset{\text{def}}{=} \text{let } x = \text{choose(true, false)} \text{ in if } x = \text{true} \text{ then } P \text{ else } Q
\]

We then have \( P + Q \rightsquigarrow^2 P \) and \( P + Q \rightsquigarrow^2 Q \).
We use this notion of computation — which correctly reflects (some) real-world computations — to give a first formal security definition.

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

- terms in \( A \) belong to \( T_{\text{pub}}(N) \) and \( \text{fn}(P, s) \cap \text{fn}(A) = \emptyset \),
- \( s =_{E} s' \) and
- \( P \parallel A \xrightarrow{s, s'} Q \) and names chosen in this reduction when reducing 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 4.** For each of the following processes, indicate when the secrecy of \( n \) is ensured, and exhibit an adversary otherwise:

- \( P_1 = \text{new } k. \text{out}(c, \text{senc}(n, k)). \text{out}(c, k) \)
- \( P_2 = \text{in}(c, x). \text{out}(c, \text{senc}(n, x)) \)
- \( P_3 = \text{out}(c, \text{senc}(n, k)). \text{in}(c, x) \cdot \text{if } x = n \text{ then } \text{out}(c, k) \)
- \( P_4 = \text{in}(c, x). \text{let } y = \text{adec}(x, k) \text{ in } \text{out}(c, k) \text{ else } \text{out}(c, \text{aenc}(n, \text{pk}(k))) \)
- \( P_5 = \text{!P}_4 \)

**Exercise 5.** Show that the condition of definition \( \square \) on the choice of fresh names in the reduction is necessary, by providing an undesirable example that would count as a breach of secrecy without the condition. Similarly, show the importance of \( \text{fn}(s) \cap \text{fn}(A) = \emptyset \), i.e. show that the definition would not adequately model secrecy if \( \text{fn}(P) \) were used instead of \( \text{fn}(P, s) \) in both places. Conversely, you may finally show that removing the condition \( s =_{E} s' \) would yield an equivalent definition.

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

**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.
\[(\text{out}(c,u).P \mid Q, \Phi) \xrightarrow{\text{out}(c,w)} (P \mid Q, \Phi \cup \{w \mapsto u\}) \quad \text{when } w \notin \text{dom}(\Phi)\]

\[(\text{in}(c,x).P \mid Q, \Phi) \xrightarrow{\text{in}(c,R)} (P\{x \mapsto u\} \mid Q, \Phi) \quad \text{when } R \in T_{\text{pub}}(N \cup \text{dom}(\Phi)), R \notin \text{bn}(\Phi) \text{ and } R\Phi \Downarrow u\]

\[(\text{let } x = t \text{ in } P \text{ else } Q \mid R, \Phi) \xrightarrow{\ell} (P\{x \mapsto u\}, \Phi) \quad \text{when } t \Downarrow u\]

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

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

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

Figure 1.1: Labelled transitions between configurations

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 \(x \dashv y\) 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, \(R \dashv (P, \Phi)\) means that \(R \in T_{\text{pub}}(W \cup N \setminus \text{fn}(P, \Phi))\).

**Definition 5 (Labelled transitions).** The labelled transition relation \(K \leadsto K'\), given by the rules of fig. [1.1] 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 \(\leadsto\) as follows: \(K_0 \leadsto K_n\) when \(\tau = \alpha_1 \ldots \alpha_n\) and \(K_i \leadsto K_{i+1}\) for all \(i \in [1, n]\).

As for the internal reduction, the choices of fresh names in labelled transitions is irrelevant. Formally, we have the following analogue of proposition [1].

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

The main novelty with the labelled transition system 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 proposition [2] below.

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 \(\tau, P', \Phi'\) such that \((P, \text{fn}(P, s), \emptyset) \leadsto (P', \Phi')\) and \(\Phi' \vdash s\).
Proof. We first prove that, if there exists an execution \((P, \Phi) \xrightarrow{u} (P', \Phi')\) 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 \(P \mid A \Phi \xrightarrow{*} \text{out}(\ldots, s)\) \(\ldots\) 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 \(tr = \epsilon\) 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 \(A \Phi \xrightarrow{\epsilon} \text{out}(c, s)\).

Assume now that \(tr = \alpha.\text{tr}'\). We have \((P, \Phi) \xrightarrow{\alpha \cdot \text{tr}'} (P', \Phi')\) 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 \(P \mid A \Phi \xrightarrow{\epsilon} P_1 \mid A_1 \Phi_1\).

- If \(\alpha = \text{in}(c, R)\) we have \(R \notin \text{bn}(\Phi), R \Phi \Downarrow u\) and

\[(P, \Phi) = (\text{in}(c,x).Q \mid R, \Phi) \xrightarrow{\alpha} (Q\{x \mapsto u\} \mid R, \Phi) = (P_1, \Phi_1).\]

We obtain by induction hypothesis an adversary \(A_1\) against \(P_1\). The attacker \(A := \text{let } x = R \text{ in } \text{out}(c,x).A_1\) (for some \(x \notin \text{fv}(A_1)\)) allows us to conclude. We easily check that it executes well, and that it contains the same free variables as \(A_1\). It also satisfies \(A \notin \text{bn}(\Phi)\) since \(R \notin \text{bn}(\Phi)\).

- 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 \(m \notin (P, \Phi, \text{bn}(\Phi))\). By induction hypothesis we obtain an adversary \(A_1\) against \(P_1\), satisfying \(A_1 \notin \text{bn}(\Phi_1)\) and hence \(A_1 \notin \text{bn}(\Phi)\) since \(\Phi_1 = m.\Phi\). Since \(m \notin \text{fn}(A_1)\), we can reduce \(P \mid A_1 \Phi \xrightarrow{\epsilon} P_1 \mid A_1 \Phi_1\), which allows us to conclude with \(A := A_1\).

- 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 (definition \[2\]). The problem is the notion of adversary: the condition \(A \notin P\) 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 \(A \notin P\) and all terms in \(A\) belong to \(\mathcal{T}_{\text{pub}}(\mathcal{N} \cup W)\). 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, for all decorated subterms \(u^R\) occurring in it, we have \(R \in \mathcal{T}_{\text{pub}}(\text{dom}(\Phi) \cup \mathcal{N} \setminus \text{bn}(\Phi))\) and \(R \Phi \Downarrow u\). Subterms that are not decorated are required to belong to \(\mathcal{T}_{\text{pub}}(\mathcal{N} \setminus \text{bn}(\Phi))\). 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. \(A \notin \text{bn}(\Phi)\) means \(R(A) \notin \text{bn}(\Phi)\). Intuitively, we do not consider the computed terms but only the recipes that have been used to obtain them.

One can check that the existence of an adversary in the sense of definition \[2\] is equivalent to the existence of a \(\Phi\)-adversary for \(\Phi = \text{fn}(P, s)\). Thus, it suffices to establish that

\[\text{In particular, a } \Phi\text{-adversary where all decorated terms are of the form } \Phi(u)^w\text{ is simply a process of the form } A \Phi.\]
if there exists a $\Phi$-adversary $A$ such that $P \mid A \rightsquigarrow^* \text{out}(\_, s') \mid \ldots$ for some $s' \equiv_{E} s$ with a reduction that does not pick fresh names in $\text{bn}(\Phi)$,

then there exists a trace $(P, \Phi) \xrightarrow{\tau} (P', \Phi')$ such that $\Phi' \vdash s$.

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 $\text{tr} = \text{out}(c, w)$ for some fresh $w$ since $\Phi \cup \{ w \mapsto s' \} \vdash s$ by taking $R = w$; if it belongs to $A$ we have $R(s')\Phi \Downarrow s$ with $R(s') \notin \text{bn}(\Phi)$, hence $\Phi \vdash s$ so we can conclude with $\text{tr} = \epsilon$.

If the first reduction step is internal to $A$, i.e. $P \mid A \rightsquigarrow P' \mid A'$ with $A \rightsquigarrow A'$, we conclude by induction hypothesis on the rest of the reduction involving $A'$, keeping $\text{tr}$ unchanged. If the reduction is a name creation, we have assumed that the chosen name satisfies $m \notin \text{bn}(\Phi)$, hence $A' \notin \text{bn}(\Phi)$. 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 $R(t)\Phi \Downarrow u$.

Assume now that the first step is a communication occurring inside $P$, exchanging $u$ on channel $c$. We have

$$(P, \Phi) \rightarrow_{\text{out}(c, w), \text{in}(c, w)} (P', \Phi \cup \{ w \mapsto u \})$$

which allows us to conclude by induction hypothesis with $A$, which is also a $(\Phi \cup \{ w \mapsto u \})$-adversary against $P'$.\[\Box\] If $t$ occurs in a $\Phi$-adversary and $t \Downarrow u$, then $R(t)\Phi \Downarrow u$.

For other reductions internal to $P$ we conclude by induction hypothesis with $\text{tr} = \tau.\text{tr}'$. If $P$ creates a name $m$, we need $A$ to be a $(m, \Phi)$-adversary but we do have $A \notin \text{bn}(m, \Phi)$ by the freshness condition on $m$ in the internal reduction semantics, and because $A \notin \text{bn}(\Phi)$.

Assume that, in the first reduction step, a message $u$ is sent by $A$ and received by $P$ on some channel $c$. Since $A \notin \text{bn}(\Phi)$ we have $R(u) \notin \text{bn}(\Phi)$ and thus

$$(P, \Phi) \rightarrow_{\text{in}(c, R(u))} (P' \{ x \mapsto u \}, \Phi).$$

We conclude by induction hypothesis using $P' \{ x \mapsto u \}$ and $A'$: one easily checks that $A'$ is still a $\Phi$-adversary, in particular $A' \notin \text{bn}(\Phi)$.

Consider finally the case where $P$ sends some message $u$ to $A$. Then

$$(P, \Phi) \rightarrow_{\text{out}(c, u)} (P', \Phi \cup \{ w \mapsto u \})$$

and we conclude by induction hypothesis using $P'$ and $A' \{ x \mapsto u^w \}$. \[\Box\]

**Exercise 6.** For each process $P$ of exercise 4 exhibit (if it exists) an execution $(P, \emptyset) \rightarrow_{\tau} (Q, \Phi)$ and a recipe $R \notin \text{bn}(\Phi)$ such that $R\Phi \Downarrow n$.  

11
Chapter 2

Verifying secrecy for bounded executions

In this section we will present an important approach to verify secrecy, based on (forward) 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 precision, 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 reachable 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:

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

2.1 Deciding deduction

In the concrete semantics of fig. 1.1 we have been using recipes to witness how an attacker might derive a given message. However, when checking reachability properties 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. 2.1 for the primitives considered here. The first rule is called the axiom rule, then there is, for each constructor $f \in \Sigma_c = \{\text{aenc}, \text{pair}\}$, 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 $f$.

**Proposition 4.** Let $\Phi$ be a frame. We have $\Phi \vdash u$ (in the sense of definition 6) iff $\Phi \vdash R : u$ is derivable (in the sense of fig. 2.1).

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. 2.1 can be simplified to
Φ(φ) = u
Φ ⊢ w : u
Φ ⊢ n : n
Φ ⊢ R : u
Φ ⊢ pk(R) : pk(u)
Φ ⊢ pair(R₁, R₂) : pair(u₁, u₂)
Φ ⊢ proj₁(R) : u₁
Φ ⊢ aenc(R₁, R₂) : aenc(u₁, u₂)
Φ ⊢ adec(R₁, R₂) : u
Φ ⊢ R1 : u₁
Φ ⊢ R2 : u₂
Φ ⊢ R1 : aenc(u₁, R₁)
Φ ⊢ R2 : u
Φ ⊢ u

Φ(φ) = u
Φ ⊢ w : u
Φ ⊢ n : n
Φ ⊢ R : u
Φ ⊢ pk(R) : pk(u)
Φ ⊢ pair(R₁, R₂) : pair(u₁, u₂)
Φ ⊢ proj₁(R) : u₁
Φ ⊢ aenc(R₁, R₂) : aenc(u₁, u₂)
Φ ⊢ adec(R₁, R₂) : u
Φ ⊢ R1 : u₁
Φ ⊢ R2 : u₂
Φ ⊢ R1 : aenc(u₁, R₁)
Φ ⊢ R2 : u
Φ ⊢ u

Figure 2.1: Decorated deduction rules.

omitting recipes. Instead of deriving judgments of the form Φ ⊢ R : u, judgments are simply of the form Φ ⊢ u and each rule is then adapted in the straightforward way. For example, the first two rules become:

\[ \frac{u \in T}{\Phi \vdash u} \quad \frac{n \in N \setminus \text{bn}(\Phi)}{\Phi \vdash n} \]

This does not change the deducible messages, and in fact it is easy to reconstruct the missing recipes to get back, from a derivation in this new style, a derivation in the original system.

Also observe that Φ 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 Φ ⊢ u given Φ and u is in PTIME.

**Proof sketch.** If there is a derivation of Φ ⊢ u, then there is a derivation in which no two points of a branch derive the same message. In derivations without such repetitions, a composition rule cannot be used to deduce the left premise of a judgment. Indeed, this can only in only two ways, shown next, and each involves a repetition (in the case of pairs, the left premise of the decomposition rule is its only premise):

\[ \frac{\Phi \vdash v_1 \quad \Phi \vdash v_2}{\Phi \vdash \text{pair}(v_1, v_2)} \quad \frac{\Phi \vdash v \quad \Phi \vdash \text{pk}(w)}{\Phi \vdash \text{aenc}(v, \text{pk}(w))} \]

To conclude, it suffices to observe that, when only decompositions and axioms may be used to derive the left premise of a decomposition, all statements Φ ⊢ v are such that v is either a subterm of v (the conclusion) or of Φ. More specifically, we would show by induction on a derivation Π of Φ ⊢ v that: if Π ends with an axiom or decomposition then all of its internal judgments are of the form Φ ⊢ u with u a subterm of Φ; otherwise (when Π ends with a composition rule) its internal judgments are of the form Φ ⊢ u where u is a subterm of either Φ or v.

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. 1.1. Compared to that LTS, to which we will refer as the concrete LTS, the symbolic LTS will allow free variables in its configurations. These free variables will stand for unknown messages; they will thus only be instantiated by constructor terms. 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. 2.2 and 2.3 the relations \( t \not\leq x \) and \( t \not\leq x \perp  \) where \( t \in T(N \cup X) \), \( x \in X \) and \( \varphi \) is a conjunction of equalities between terms of \( T_c(N \cup X) \). Intuitively, \( t \not\leq x \) means that
We define \(\text{Sol}(\varphi)\) as the set of all \(\theta : \text{fv}(\varphi) \to \mathcal{T}_c(\mathcal{N})\) 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 \(t \in \mathcal{T}(\mathcal{N} \cup \mathcal{X})\) and \(x \in \mathcal{X}\).

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

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

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

**Example 9.** We can derive \(\text{adec}(\text{proj}_1(x), y) \Downarrow z\) with

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

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

**Example 10.** We can derive \(\text{proj}_1(\text{pair}(x_1, \text{proj}_1(x_2))) \Downarrow y\) where \(\varphi\) is

\[
x_2 = \text{pair}(x'_2, x''_2) \land x' = \text{pair}(x_1, x'_2) \land x' = \text{pair}(y, y')
\]

up to the removal of identities over variables, which is equivalent to the condition

\[
x_2 = \text{pair}(x'_2, x''_2) \land x_1 = y.
\]

Note that the constraint imposes that \(x_2\) is a pair, even though it is ignored in the result.
Definition 7. A deducibility constraint system is a finite set of deducibility constraints of the form $T \vdash^1 u$ 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 $T_1 \vdash^1 u_1 \land \ldots \land T_n \vdash^1 u_n$:

- **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 $\theta : \text{fv}(C) \rightarrow T_c(N)$ and, for each constraint $T \vdash^1 u$ of $C$, $T\theta \vdash^1 u\theta$.

Definition 9. A symbolic configuration is a tuple $(P, \Phi, C)$ where $P$ is a process with free variables, $\Phi = \vec{\nu}.\sigma$ is a frame whose messages may contain free variables, and $C$ is a constraint system.
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 $[K_1] \Rightarrow K_2'$ and $\theta \in \text{Sol}(K_2')$, then $K_1 \Rightarrow [K_2']\theta$ for some $\theta$.
- If $K_1 \Rightarrow K_2$ then there exists $K_2'$ and $\theta \in \text{Sol}(K_2')$ such that $[K_1] \Rightarrow K_2'$ and $K_2 = [K_2']\theta$.

Moreover, $\text{tr}$ and $\text{tr}'$ only differ in input actions: when $x$ occurs in an input of $\text{tr}'$, a recipe that allows to derive $\theta(x)$ occurs in the corresponding position of $\text{tr}$.

**Example 11.** Consider the process $P := \text{in}(c, x) \cdot \text{let } y = \text{adec}(x, k) \text{ in } \text{out}(c, y)$ and the initial knowledge $\Phi := \{w \mapsto \text{pk}(k)\}$. Then we have

$$(P, \Phi, \top) \xrightarrow{\text{in}(c, x)} (P', \Phi, \top \vdash x) \xrightarrow{\text{out}(c, y), \Phi, \top \vdash \text{adec}(y, \text{pk}(k))} (0, \Phi \cup \{w' \mapsto y\}, \Phi \vdash \text{aenc}(y, \text{pk}(k))).$$

### 2.3 Solving deducibility constraint systems

We refer the reader to chapter 3 of the legacy lecture notes\(^1\) for this part. There are a few superficial differences:

- The legacy 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 legacy lecture notes feature a deduction system with judgments of the form $\vdash u$ rather than $\Phi \vdash u$. This is 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 $\text{adec}(\text{aenc}(x, \text{pk}(y)), y) \rightarrow x$, the legacy lecture 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.

- The legacy notes feature a deduction system without a name rule. Without such a rule, proposition 4 does not hold anymore. However, the modification can justified as follows: when considering a constraint system expressing some reachability or secrecy goal, we only care about the existence of solutions; if there is a solution featuring attacker-generated names, there is also one where these names are all replaced by the same arbitrary term (which exists since the first frame is assumed to be non-empty), and that solution is accounted for in the deduction system without the name rule.

- The symbolic semantics that we have described only deal with asymmetric encryption and pairs, but does not have the $\text{eq}$ destructor (cf. example 5).

\(^1\)http://www.lsv.fr/~baelde/secu/poly.pdf
which allows to encode basic conditionals! Adding them would force us to consider disequality constraints in addition to equalities and deduction constraints. This can be done, but is out of the scope of our lectures. Note that this also complexifies slightly the above discussion regarding names.
Chapter 3

Verifying secrecy for unbounded executions

The slides for this part are available online. The reference is the corresponding chapter in the old lecture notes.
Chapter 4

Equivalences

In this section we will define several equivalences on processes. The goal is to model indistinguishability by an outside observer, which in turn can be used to formally define several security properties. This style of modelling has become popular in particular for privacy-type properties, that often cannot be express as properties of traces (unlike secrecy or correspondence properties). As we shall see, several equivalences may be considered, which may be more or less realistic, and more or less difficult to verify.

4.1 Static equivalence

We start by defining an observational equivalence on frames. Intuitively, two frames are statically equivalent when an attacker will always obtain the same results when computing on one frame or the other.

**Definition 10.** Static equivalence is the least symmetric relation on frames such that two frames $\Phi$ and $\Psi$ are statically equivalent, written $\Phi \sim \Psi$ whenever all of the following conditions hold:

- $\text{dom}(\Phi) = \text{dom}(\Psi)$;
- for all $R \in \mathcal{T}(\text{dom}(\Phi) \cup \mathcal{N} \setminus \text{bn}(\Phi, \Psi))$, if there exists $u$ such that $R \Phi \downarrow u$ then there exists $v$ such that $R \Psi \downarrow v$;
- for all $M, N \in \mathcal{T}(\text{dom}(\Phi) \cup \mathcal{N} \setminus \text{bn}(\Phi, \Psi))$, if there exists $u$ and $v$ such that $M \Phi \downarrow u$, $N \Phi \downarrow v$ and $u =_E v$, then there exists $u'$ and $v'$ such that $M \Psi \downarrow u'$, $N \Psi \downarrow v'$ and $u' =_E v'$.

We may abbreviate the second condition as “$R \Phi \downarrow \iff R \Psi \downarrow$” and the third as “$M \Phi \downarrow \iff M \Psi \downarrow =_E N \Phi \downarrow \iff N \Psi \downarrow$” — recall that static equivalence is symmetric.

**Example 12.** Regardless of the available cryptographic primitives we have:

- $(n, m).\{w \mapsto n\} \sim (n, m).\{w \mapsto m\}$;
- $(n, m).\{w \mapsto n, w' \mapsto n\} \not\sim (n, m).\{w \mapsto m, w' \mapsto n\}$;
- $n.\{w \mapsto n\} \not\sim n.\{w \mapsto m\}$.

**Example 13.** Assuming that asymmetric encryption is modelled using the reduction $\text{adec}(\text{aenc}(x, \text{pk}(y)), y) \rightarrow x$ alone, we have:

- $(k, n).\{w \mapsto \text{aenc}(m, \text{pk}(k))\} \sim (k, n).\{w \mapsto n\}$.

We consider recipes that correspond to computations performed by the attacker, who does not have access to the secret/fresh names of either frame.
simple processes. Consider for example the processes of a synchronous network. Second, some reasonable attacks may not be representable by our servables. It might not be desirable, if the protocol and attacker are played on an asynchronous network, thus a process can observe when its output has been received; such observations can be important. Some subtle choices. First, our process algebra is synchronous: outputs have a constant delay.

A distinguisher is a process. If it fails, then there is a test (a distinguisher, or attacker) that can reach the success output when put in parallel with one process but fails. However, in a practical setting where process executions are reliable, an attacker could observe only with must testing. Such distinctions would be captured through may testing, and by stronger equivalences discussed below.

4.2 May testing

We define a first equivalence on processes, that takes into account the possible executions of the process, relying on internal reduction.

Definition 11. A test is a process with no free name and in which a special channel T may occur. A process P may pass a test T, written $P \models T$ if $P \mid T \xrightarrow{*} \text{out}(T, u) \mid Q$ for some $u$ and $Q$.

We define $T(P) := \{ T \mid P \models T \}$.

We say that the process may pass the test because there is one execution where the success output is reached. Alternatively, one might say that the process must pass the test when, for all (sufficiently complete or fair) traces, the success output is reached.

Definition 12. Processes $P$ and $Q$ are may-testing equivalent when $T(P) = T(Q)$.

May-testing equivalence can be related to static equivalence; the proof of the following proposition relies on the same ingredients as proposition 3.

Proposition 9. If $(P, \emptyset) \xrightarrow{\Phi} (P', \Phi')$ and $(Q, \emptyset) \xrightarrow{\Psi} (Q', \Psi')$ with $\Phi \not\approx \Psi$, then $P$ and $Q$ are not may-testing equivalent.

May-testing equivalence requires more than static equivalence for the reachable frames: it also imposes that the same actions are feasible for both processes. For instance, if $P$ can perform an output on $c$ and not $Q$, then the test $\text{in}(c, x).\text{out}(T, \text{ok})$ belongs to $T(P) \setminus T(Q)$.

May testing is arguably the most natural notion of indistinguishability, where the distinguisher is a process. If it fails, then there is a test (a distinguisher, or attacker) that can reach the success output when put in parallel with one process but not the other: the distinguisher can tell, in such a case, with whom he is interacting.

However, as often, modelling indistinguishability through may testing implies some subtle choices. First, our process algebra is synchronous: outputs have a constant delay, thus a process can observe when its output has been received; such observations might not be desirable, if the protocol and attacker are played on an asynchronous network. Second, some reasonable attacks may not be representable by our simple processes. Consider for example the processes $P := \text{out}(c, \text{ok}).\text{out}(c, \text{ok})$ and $Q := \text{out}(c, \text{ok}) + \text{out}(c, \text{ok}).\text{out}(c, \text{ok})$ (using non-deterministic choice as encoded e.g. in example 8). Both processes may pass the test $\text{in}(c, x).\text{in}(c, y).\text{out}(T, \text{ok})$. However, in a practical setting where process executions are reliable, an attacker could observe only with $Q$ a situation where only one output is emitted — note that such an attacker cannot be represented as a process in our calculus, since there is no way to tell that an output won’t happen, or hasn’t happened after some delay. Such distinctions would be captured through must testing, and by stronger equivalences discussed below.

\[ (k, n).\{ w_0 \mapsto pk(k), w \mapsto aenc(m, pk(k)) \} \not\approx (k, n).\{ w_0 \mapsto pk(k), w \mapsto n \}; \]

\[ (k, n, m).\{ w \mapsto aenc(m, pk(k)), w' \mapsto k \} \not\approx (k, n, m).\{ w \mapsto n, w' \mapsto k \}; \]

\[ (k, n, m).\{ w \mapsto aenc(m, pk(k)), w' \mapsto aenc(n, pk(k)), w_0 \mapsto pk(k) \} \not\approx (k, n, m).\{ w \mapsto aenc(m, pk(k)), w' \mapsto aenc(m, pk(k)), w_0 \mapsto pk(k) \}. \]

As seen in the accompanying slides, static equivalence can be used to formally define offline guessing attacks.
4.3 Trace equivalence

Conceptually, trace equivalence replaces the tests of may testing by traces and frames of the labelled transition system. Basically, two processes are trace equivalent if, for every execution that one process can perform in the LTS, the other process can perform an execution with the same trace (up to // actions, which are not observable) and resulting in a frame that is statically equivalent.

**Definition 13.** We write $K \approx K'$ when tr does not contain any $//$ action and there exists a trace $tr'$ obtained from tr by inserting $//$ actions such that $K \xrightarrow{u} K'$.

Trace equivalence is then often defined from $Tr(K) = \{ (tr, \Phi(K')) \mid K \approx K' \}$ as the relation $K \approx K'$ which holds when for all $(tr, \Phi) \in Tr(K)$ there exists $(tr, \Psi) \in Tr(K')$ such that $\Phi \sim \Psi$, and vice versa. We adopt an equivalent but more concise definition below, via the definition of a meaningful notion of observable $Tr'$.

**Definition 14.** When $K$ is a configuration, we define

$$Tr'(K) = \{ (tr, \Psi) \mid K \approx K', \Phi(K') \sim \Psi \}.$$  

We say that $K$ and $K'$ are trace equivalent when $Tr'(K) = \{ (tr, \Psi) \mid K \approx K', \Phi(K') \sim \Psi \}$.

Note that this definition only makes sense when $dom(\Phi(K)) = dom(\Phi(K'))$. More surprisingly, it essentially requires $bn(\Phi(K)) = bn(\Phi(K'))$, otherwise trace equivalence breaks for silly reasons. For instance, supposing that there exists a name $n \in bn(\Phi(K')) \setminus bn(\Phi(K))$, $K$ admits traces that use $n$ in recipes, whereas this would be forbidden for $K'$. This condition on bound names should be ensured first by $\alpha$-converting one of the two configurations. Anyway, we tend to use trace equivalence on processes, i.e. configurations with empty frames: when $P$ and $Q$ are processes, we write $P \approx Q$ when $(P, \emptyset) \approx (Q, \emptyset)$.

As we shall see, trace equivalence is close to may testing, but the two notions only coincide under (mild) conditions.

**Proposition 10.** Assume that computation is deterministic in the following sense: for all $t$, $u$ and $v$ such that $t \Downarrow u$, $t \Downarrow v$ and $u \equiv_E v$ are equivalent. Then trace equivalence implies may-testing equivalence.

We first show that the determinism assumption is necessary.

**Example 14.** Consider $\Sigma = \{ f, g, ch, val \}$ with $\Sigma_{pub} = \{ ch, val \}$. The unary symbols $f$ and $g$ are private constructors, and the equational theory $E$ is empty. The public constructors $ch$ (unary) and $val$ (binary) are associated with the following reductions:

\[ ch(x) \rightarrow f(x) \quad val(x, f(x)) \rightarrow x \]
\[ ch(x) \rightarrow g(x) \quad val(x, g(x)) \rightarrow x \]

Consider now two processes:

\[ P := \text{new } n. \text{out}(c, n).\text{in}(c, x).\text{in}(c, y). \]
\[ \text{let } _= \text{val}(n, x) \text{ in let } _= \text{val}(n, y) \text{ in if } x = y \text{ then } \text{out}(c, n) \text{ else } 0 \]
\[ Q := \text{new } n. \text{out}(c, n).\text{in}(c, x).\text{in}(c, y). \]
\[ \text{let } _= \text{val}(n, x) \text{ in let } _= \text{val}(n, y) \text{ in if } x = y \text{ then } 0 \text{ else } \text{out}(c, n) \]

It is easy to see that $T(P) \neq T(Q)$. Specifically, the following test passes only with $P$:

\[ \text{in}(c, z).\text{let } z' = \text{ch}(z) \text{ in out}(c, z').\text{out}(c, z').\text{in}(c, _).\text{out}(\_ , _).0 \]

Let us now show that $Tr'(P) = Tr'(Q)$. It is easy to see that these sets coincide for traces of length up to 4. More precisely, they both contain exactly the prefixes of
out\((c,w),\) in\((c,R),\) in\((c,R')\). \(\tau\) (for arbitrary \(w, R\) and \(R'\)) together with any frame \(\Phi \sim n,\{w \mapsto n\}\). In order to execute one more \(\tau\) action, we must have \(R\Phi \updownarrow f(n)\) or \(R\Phi \updownarrow g(n)\): since \(f\) and \(g\) are private, we must have \(R = \text{ch}(w)\), regardless of whether we are considering \(P\) or \(Q\). In order to execute a third \(\tau\) action, we must similarly have \(R' = \text{ch}(w)\). Hence Tr\((P)\) and Tr\((Q)\) coincide for traces of length up to six.

The only trace of length seven that could be considered is
\[
\text{out}(c,w),\text{in}(c,\text{ch}(w)),\text{in}(c,\text{ch}(w)).\tau.\tau.\tau.\text{out}(c,w')
\]

(this is technically a choice in \(w\) and \(w'\) but it does not change the argument). This trace can be executed by \(P\) and \(Q\), with resulting frames \(\Phi \sim n,\{w' \mapsto n\}\): crucially, for \(Q\), this is because it is possible to take \(\text{ch}(w)\).

Proof of proposition[70] Under the computation determinism assumption we show that, when \(P_0 \approx Q_0\), any test \(T_0\in T(P)\) also belongs to \(T(Q)\). From \(P_0 | T_0 \rightarrow^* \text{out}(T,\_ | \_\) we obtain a reduction
\[
P_0 | T_0\theta_0 \rightarrow^* P_0' | T_0'\theta_0' \rightarrow P_1 | T_1\theta_1 \rightarrow^* P_1' | T_1'\theta_1' \ldots \rightarrow P_n | T_n\theta_n \rightarrow^* P_n' | T_n'\theta_n'
\]
such that:

1. \((P,\emptyset) = (P_0,\Phi_0) \not\Rightarrow (P_1,\Phi_1) \ldots \not\Rightarrow (P_n,\Phi_n)
2. for every \(x \in \text{dom}(\theta_i')\) there exists \(R_x\) such that \(R_x\Phi_n \downarrow \theta_i'(x)\)
3. for every \(i, T_i\theta_1 \rightarrow^* T_i\theta_i'\) and \(\theta_i \subseteq \theta_i' \subseteq \theta_{i+1}\)
4. for every \(i, (T_i, T_i') \not\updownarrow \text{bn}(\Phi_i)\)
5. \(T_n' = \text{out}(T,\_ | \_\)
6. for every \(i\) there exists a conjunction of equalities \(\varphi_i\) such that for all \(\sigma, \models \varphi_i\sigma \text{ iff } T_i\sigma \rightarrow^* T_i'\sigma\)

Here, \(\models \varphi\sigma\) means that \(\varphi\) consists only of identities modulo the equational theory. Note that condition 2 is expressed only for \(\theta_n'\) but actually applies to all \(\theta_i\) and \(\theta_i'\) by condition 3.

Conditions 1 and 3 are obtained by decomposing the original reduction into portions where \(P_i\) and \(T_i\theta_i\) reduce separately followed by one communication between the two.

- The internal reductions of \(T_i\theta_i\) are mapped to \(T_i\theta_i \rightarrow^* T_i\theta_i'\). There we ensure wlog that fresh names are picked outside of \(\text{bn}(\Phi_i)\). When let constructs are evaluated, we can construct \(T_i'\) satisfying 4: the key case is when \(T_i\theta_i = \text{let } x = t\theta_i \text{ in } P\theta_i\). else \(Q\theta_i\) reduces to \(P\theta_i\{x \mapsto u\}\), for which we set \(T_i' = P\) and extend \(\theta_i\) with \(\{x \mapsto u\}\), for which there exists a recipe \(R_x\) satisfying condition 2, obtained by replacing in \(t\) any variable \(y\) by its associated recipe \(R_y\). Still concerning let evaluations, we extract a formula \(\varphi_i\) that encapsulates the conditions on \(T_i\theta_i\) that are sufficient and necessary for the corresponding reduction to happen, so that condition 6 above is satisfied. (This is essentially what we did in the symbolic semantics with the \(\updownarrow \varphi\perp x\) and \(\updownarrow \varphi\perp \perp\) predicates of figs. 2.2 and 2.3)

- The internal reductions \(P_i \rightarrow^* P_i'\) are mapped to \(\tau\) transitions performed implicitly before the \(\alpha_{i+1}\) action of \((P_i,\Phi_i) \not\updownarrow \uparrow (P_{i+1},\Phi_{i+1})\).

- When the communication is an output from \(P_i\), we have \(\alpha_{i+1} = \text{out}(c,u)\) for some \(c\) and \(u\). We have \(\theta_{i+1} = \Phi_i \cup \{w \mapsto u\}\) for some new handle \(w\). The process \(T_i'\) is of the form \(\text{in}(c,x),Q | R\) and we set \(T_{i+1}' = Q | R\) with \(\theta_{i+1} = \theta_i' \cup \{x \mapsto u\}\). This extension satisfies condition 2 by taking \(R_x = w\).
• When $P_i$ performs an input, $T_i' = \text{out}(c,u).Q \mid R$ and the recipe $R_u = u(x \mapsto R_u)_{x \in \text{dom}(u)}$ is such that $R_u \Phi_i \Downarrow u$ by condition 2; we thus take $\alpha_{i+1} = \text{in}(c,R_u)$.

By $P \approx Q$ we have $(\alpha_1 \ldots \alpha_n, \Psi_n) \in \text{Tr}(Q)$ for some frame $\Psi_n$ such that $\Phi_n \sim \Psi_n$. Moreover we can choose $\Psi_n$ wlog to obtain $(T_i, T_i') \not\approx \text{bn}(\Psi_n)$ for all $i$. Consider now $\varphi := \land_i \varphi_i$; we have $\models \varphi = \models \varphi_i$ for some $\sigma_n$ such that, for every $x \in \text{dom}(\sigma_n) = \text{dom}(\theta_n^0)$, $R_u \Psi_n \Downarrow \sigma_n(x)$. From $\sigma_n$ we derive, for all $i$, $\sigma_i = \sigma_n|_{\text{dom}(\theta_i)}$ and $\sigma_i' = \sigma_n|_{\text{dom}(\theta_i')}$: By condition 6 we obtain that, for all $i$, $T_i \sigma_i' \approx^* T_i' \sigma_i'$. The execution $(Q,\emptyset) = (Q_0, \Psi_0) \rightarrow^* (Q_n, \Psi_n)$ yields $Q_i \approx^* Q_i'$ for all $i$, with $(Q_i', \Psi_i) \rightarrow^+ (Q_{i+1}, \Psi_{i+1})$. We finally check that $Q_i' \mid T_i' \sigma_i' \approx Q_{i+1} \mid T_{i+1} \sigma_{i+1}$:

• If $\alpha_{i+1} = \text{in}(c,R_u)$ we have an output $\text{out}(c,u)$ at toplevel in $T_i'$, with $R_u \Phi_i \Downarrow u \theta_i'. \Downarrow$. Then $R_u \Psi_i \Downarrow u \sigma_i'$. Since $Q_i'$ perform $\alpha_{i+1}$ in the LTS, we have $Q_i' = \text{in}(c,x).Q \mid R$ and $Q_{i+1} = Q\{x \mapsto v\} \mid R$ for some $v$ such that $R_u \Psi_n \Downarrow v$. By determinism of $\Downarrow$, we obtain that $u \sigma_i' = v$. Since our semantics identify terms modulo $E$, we can assume wlog that $u \sigma_i' = v$, which allows to conclude. 

### 4.4 Observational equivalence and bisimulation

Observational equivalence and bisimulation are two similar notions, the former being defined in terms of internal reductions, the latter in terms of labelled transitions. Both are more constraining (finer) than may testing or trace equivalence, as they require related processes to mimic each other step by step.

In order to define observational equivalence we define a basic notion of observable, sometimes called a “barb”. Arbitrarily, we only consider outputs as observables: we write $P \Downarrow c$ when $P$ can output on $c$ after internal reductions, i.e. $P \rightarrow^* \text{out}(c,u).P' \mid P''$.

**Definition 15.** The binary relation $\mathcal{R}$ over closed processes is an observational bisimulation if it is symmetric and $P \mathcal{R} Q$ implies:

- for all $c$, $P \Downarrow c$ implies $Q \Downarrow c$;
- for all $P'$, $P \rightarrow^* P'$ implies $Q \rightarrow^* \mathcal{R} P'$;
- for all $R$, $(P \mid R) \mathcal{R} (Q \mid R)$.

Observational equivalence is the largest observational bisimulation.

Note that the largest observational bisimulation is well-defined, as the union of observational bisimulations is still an observational bisimulation.

The first clause requires that related processes exhibit the same observables. The last one requires that equivalence is preserved when including processes in larger contexts: this is a nice and strong property that corresponds well to the idea that we do not want any process to be able to distinguish two equivalent processes (by interacting in parallel with them). The particularity of observational equivalence (compared, for instance, with may testing) lies in the second clause which requires that equivalence is preserved step by step through internal reductions.

**Example 15.** The processes $\text{in}(c,x).0$ and $0$ are not observationally equivalent. Assume there is an observational equivalence relating them. Then the third clause forces us to also relate $\text{out}(c,\text{ok}) \mid \text{in}(c,x).0$ and $\text{out}(c,\text{ok}) \mid 0$, that is $\text{out}(c,\text{ok})$. Now the second clause requires to relate $0$ and a reduct of $\text{out}(c,\text{ok})$, which can only be $\text{out}(c,\text{ok})$ itself. But the first clause forbids us to relate these two processes since only the first one exhibits an output on $c$. 

\n
[23]
Example 16. Let us write $\alpha$ for the output prefix $\text{out}(c_{\alpha}, \text{ok})$, and similarly for $\beta$ and $\gamma$. We do not have observational equivalence between $\alpha.(\beta + \gamma)$ and $\alpha.\beta + \alpha.\gamma$, although these processes are may-testing and trace equivalent. This is because of the second clause: we have $\alpha.\beta + \alpha.\gamma \leadsto \alpha.\beta$, and no possible reduction on the other side, so an observational bisimulation would have to relate $\alpha.\beta$ and $\alpha.(\beta + \gamma)$. But that is impossible: when put in parallel with $\text{in}(c_{\alpha}, x).0$, the second process can reduce (in two steps) to $\gamma$, exhibiting an output on $c_{\gamma}$, which the first process cannot do.

We now turn to the analogue notion using labelled transitions.

Definition 16. The binary relation $\mathcal{R}$ over configurations is a bisimulation if it is symmetric and $A \mathcal{R} B$ implies:

- $\Phi(A) \sim \Phi(B)$;
- $A \xrightarrow{\tau} A'$ implies $B \xrightarrow{\tau} \mathcal{R} A'$;
- $A \xrightarrow{\tau} A'$ implies $B \xrightarrow{\tau} \mathcal{R} A'$.

Bisimilarity is the largest bisimulation.

Abadi, Blanchet and Fournet have shown that observational equivalence and bisimilarity coincide for the original applied pi-calculus, which is mostly a superset of the language considered here, except for the fact that it does not feature destructors but only an equational theory over terms.

While verifying observational equivalence is difficult due to the quantification over all contexts in its third clause, verifying bisimilarity is much simpler: one can simply attempt to construct a bisimulation containing the desired processes, and add more processes to that relation as needed. Of course, this is still not easy: the bisimulation relations are rarely finite, and can be tricky to find.


Proof. If $A \mathcal{R} B$ and $A \xrightarrow{\tau} A'$ then $B \xrightarrow{\tau} \mathcal{R} A'$ by the second condition on $\mathcal{R}$, with $\Phi(A') \sim \Phi(B')$ by the first condition. □

Definition 17. A configuration $A$ is determinate when, for all $A \xrightarrow{\tau} A'$, and for any observable action $\alpha$, $A' \xrightarrow{\alpha} A_1$ and $A' \xrightarrow{\alpha} A_2$ imply $A_1 = A_2$.

Note that determinacy does not mean that $A \xrightarrow{\tau} A_1$ and $A \xrightarrow{\tau} A_2$ imply $A_1 = A_2$ — the hidden $\tau$ actions can add many (inessential) variations between $A_1$ and $A_2$. However it can be shown that $A \xrightarrow{\tau} A_1$ and $A \xrightarrow{\tau} A_2$ imply $A_1 \xrightarrow{\tau} A'$ and $A_2 \xrightarrow{\tau} A'$, provided the two original traces choose the same names for the same new constructs.

Proposition 12. Trace equivalence implies bisimilarity for determinate configurations.

This result is incorrect in presence of non-deterministic choice where the choice is performed through a $\tau$ action (and, a fortiori, non-deterministic computations) since $(\alpha + \beta) + \gamma \approx \alpha + (\beta + \gamma)$, and these two processes are determinate, but $(\alpha + \beta)$ is not bisimilar to any reduct of $\alpha + (\beta + \gamma)$.

Proof. Let $\mathcal{R}$ be the relation such that $A \mathcal{R} B$ iff $A \approx B$ and both $A$ and $B$ are determinate. We show that $\mathcal{R}$ is a bisimulation.

The first condition is obvious: $A \approx B$ implies $\Phi(A) \sim \Phi(B)$.

For the second condition, assume $A \xrightarrow{\tau} \tau A'$. In that case, one can show that $A' \approx A$. Thus $A' \approx B$ and $A'$ is still determinate, thus we have $A' \mathcal{R} B'$ as required by choosing $B' := B$.

\[1\] The possible $\tau$ actions do not change the frame’s contents. Moreover, consider a trace $\text{tr}$ of $A$, we can anticipate some of its $\tau$ actions, and add some more, to obtain a trace $\text{tr}$ of $A'$ resulting in the same frame. The converse holds because there is no non-deterministic $\tau$. 

24
For the last condition, since we have already seen how to mimic $\tau$ actions in the previous paragraph, we can focus on showing that $A \xrightarrow{\tau} A'$ implies $B \xrightarrow{\tau} B'$ with $A' \approx B'$. By $A \approx B$, there actually exists $B'$ such that $B \xrightarrow{\tau} B'$. Let us now show that $A' \approx B'$. Assuming $A' \xrightarrow{\tau} A''$, we have $A \xrightarrow{\tau} A''$ and thus $B \xrightarrow{\tau, \alpha}, B'_1 \xrightarrow{\tau} B'' \sim A''$. By the remarks above, since $B$ is determinate, we have $B' \approx B'_1$. Hence we also have $B' \xrightarrow{\tau} A''$ as expected. The converse argument is analogue.

4.5 Concluding remarks

We refer the reader to the slides for how security notions such as resistance against offline guessing, strong secrecy, anonymity, unlinkability, etc. are encoded as equivalences.
Chapter 5

Computational security

In this upcoming chapter we will deal with a symbolic approach \[^2\] that yields proofs of security against all computational attackers.
Bibliography
