Formal Proofs of Security Protocols

David Baelde, ENS Paris-Saclay

February 3, 2019 (work in progress)

Contents

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

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.

Then, we assume a *signature* Σ, that is a set of *function symbols* together with an arity $\mathsf{ar}_\Sigma : \Sigma \to \mathbb{N}$. Given a set of basic terms B, the set $\mathcal{T}(B)$ of *terms* generated from B using Σ 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* $\Sigma = \{\ \mathbf{senc},\ \mathbf{sdec},\ \mathbf{pair},\ \mathbf{proj}_1,\ \mathbf{proj}_2,\ \mathbf{ok}\ \}$. *The symbols* **senc** *and* **sdec***, of arity 2, represent symmetric encryption and decryption. Pairing is modeled using* **pair** *of arity 2 and projection functions* $\mathbf{proj}_1$ *and* $\mathbf{proj}_2$*, both of arity 1. Finally, the symbol* **ok** *is of arity* 0*, i.e. it is a constant.*

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

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

Given a term t, we define $\mathsf{fv}(t)$ as the set of variables that occur in t. Similarly, $\mathsf{fn}(t)$ 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 $\mathcal{X}$ to $\mathcal{T}(B)$ for some B. Substitutions will be denoted by θ or σ, their domain will be noted $\mathsf{dom}(\cdot)$. 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 θ to a term t is defined as usual and noted $t\theta$. In particular, when $\mathsf{dom}(\theta) \cap \mathsf{fv}(t) = \emptyset$, $t\theta = t$.

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 $\mathbf{proj}_1(\mathbf{pair}(s, t))$ and s. Second, we provide our term algebra with a means to describe *computations* that may fail. For example, we may have that $\mathbf{sdec}(\mathbf{senc}(s, k), k)$ reduces to s but $\mathbf{sdec}(\mathbf{ok}, k)$ 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 $\mathcal{T}_c(B)$ for terms built from B using only constructor symbols, i.e., elements of Σ_c. Elements of $\mathcal{T}_c(\mathcal{N})$ 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 $\mathsf{E} \subseteq \mathcal{T}_c(\mathcal{X})^2$;

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

we will use an infix notation for it, writing $s \mathsf{E}\ t$ rather than $(s, t) \in \mathsf{E}$. Then, for any set of basic terms B we define the binary relation $=^B_{\mathsf{E}}$ over $\mathcal{T}(B \cup \mathcal{X})$ as the least equivalence relation that contains E and is closed under substitution and context closure. In other words, we impose that:

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

ϟ 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.

Example 2. *With the signature of ex. 1, and assuming that* $\Sigma = \Sigma_c$, *consider* E *made of three equations:*

$$\mathbf{sdec}(\mathbf{senc}(x, y), y) \mathsf{E}\ x, \quad \textit{and} \quad \mathbf{proj}_i(\mathbf{pair}(x_1, x_2)) \mathsf{E}\ x_i \quad \textit{for}\ i \in \{1, 2\}.$$

We then have $\mathbf{proj}_1(\mathbf{sdec}(\mathbf{senc}(\mathbf{pair}(\mathbf{ok}, n), k), k)) =^{\mathcal{N}}_{\mathsf{E}} \mathbf{ok}$ *but* $\mathbf{ok} \neq^{\mathcal{N}}_{\mathsf{E}} \mathbf{pair}(\mathbf{ok}, n)$.

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

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

Exercise 2. *Let* u *and* v *be terms such that* $u =_{\mathsf{E}} v$, *and let* x *be a variable. Show that* $t[u/x] =_{\mathsf{E}} t[v/x]$ *for any term* t. *Conclude that the* substitution principle *holds: for any terms* s, t, u, v *and variable* x, $s =_{\mathsf{E}} t$ *and* $u =_{\mathsf{E}} v$ *imply* $s[u/x] =_{\mathsf{E}} t[v/x]$.

1.1.2 Rewrite rules

We assume, for each destructor symbol $f \in \Sigma_d$ of arity n, a set of *reduction rules* of the form $f(u_1, \ldots, u_n) \to u$ where $u, u_1, \ldots, u_n \in \mathcal{T}_c(\mathcal{X})$. From this we define a *computation relation* $\Downarrow \subseteq \mathcal{T}(\mathcal{N}) \times \mathcal{T}_c(\mathcal{N})$ 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 $\mathsf{ar}(f) = n$,
  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, …, t_n and $\theta : \mathcal{X} \to \mathcal{T}_c(\mathcal{N})$ such that $t_i \Downarrow u_i\theta$ for each $i \in [1; n]$,
  $f(t_1, \ldots, t_n) \Downarrow u\theta$;
- for all t, u and v such that $t \Downarrow u$ and $u =_{\mathsf{E}} v$, $t \Downarrow v$.

ϟ 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 $=_{\mathsf{E}}$) or computable.

We write $t \not\Downarrow$ when there is no message u such that $t \Downarrow u$.

Example 3. *Consider again* Σ *from ex. 1 but assuming now that* $\Sigma_d = \{\mathbf{proj}_1, \mathbf{proj}_2\}$. *Assume that* E *contains only the equation* $\mathbf{sdec}(\mathbf{senc}(x, y), y)\ \mathsf{E}\ x$, *and take the reduction rules*

$$\mathbf{proj}_1(\mathbf{pair}(x, y)) \to x \quad \textit{and} \quad \mathbf{proj}_2(\mathbf{pair}(x, y)) \to y.$$

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

$$\mathbf{proj}_1(\mathbf{sdec}(\mathbf{senc}(\mathbf{pair}(\mathbf{ok}, n), k), k)) \Downarrow \mathbf{ok}.$$

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

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.

ϟ Note that computation failures can often be detected through equations: for instance, under the equational theory of ex. 2, if t is a message, $\langle \mathbf{proj}_1(t), \mathbf{proj}_2(t)\rangle =_{\mathsf{E}} t$ holds iff t is of the form $\langle t_1, t_2\rangle$, i.e. iff $\mathbf{proj}_1(t)$ computes successfully.

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* **pub***, and the following reduction rule:*

$$\mathbf{adec}(\mathbf{aenc}(x, \mathbf{pub}(y)), y) \rightarrow x$$

Example 5. *In several settings, it is required that destructors are deterministic up-to the equational theory, so that* $t \Downarrow u$ *and* $t \Downarrow v$ *iff* $u =_{\mathsf{E}} v$*. This is not forced by the definitions adopted in these lecture notes. For instance, we can define a typical non-deterministic binary destructor* **choose** *by the following two reductions:*

$$\begin{array}{lcl} \mathbf{choose}(x, y) & \rightarrow & x \\ \mathbf{choose}(x, y) & \rightarrow & y \end{array}$$

Example 6. *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* $\mathbf{eq} \in \Sigma_d$ *we may consider the following* list *of rules for it:*

$$\begin{array}{lcl} \mathbf{eq}(x, x) & \rightarrow & \mathbf{true} \\ \mathbf{eq}(x, y) & \rightarrow & \mathbf{false} \end{array}$$

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

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 $t = \mathbf{pair}(m, p)$, then $t\theta = \mathbf{pair}(n, n)$. As shown in that example, a renaming may not be bijective.

Exercise 3. *Assume that* $s =_{\mathsf{E}} t$ *for some* $s, t \in \mathcal{T}(\mathcal{N})$*. Show that* $s\sigma =_{\mathsf{E}} t\sigma$ *for any renaming* σ*.*

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* σ*? 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 $\mathcal{C}$ of channels, whose elements will be denoted by c, d, etc. Processes are generated from the following grammar:

$$\begin{array}{rcl|l|l} P, Q & ::= & 0 & (P \mid Q) & !P \\ & \mid & \textbf{in}(c, x).P & \textbf{out}(c, u).P & \textbf{new}\ n.P \\ & \mid & \textbf{let}\ x = t\ \textbf{in}\ P\ \textbf{else}\ Q & & \end{array}$$

where $c \in \mathcal{C}$, $x \in \mathcal{X}$, $n \in \mathcal{N}$, $u \in \mathcal{T}_c(\mathcal{N} \cup \mathcal{X})$ 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 $(P \mid P \mid P \mid \ldots)$.
- **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.
- **out**$(c, u).P$ outputs a message u on c and then behaves as P.
- **new** $n.P$ creates a new (previously unused) name m and then behaves as P with n replaced by m.
- **let** $x = y$ **in** P **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. **out**(c, u) instead of **out**$(c, u).0$ or (**let** $x = t$ **in** P) instead of (**let** $x = t$ **in** P **else** Q). When $u, v \in \mathcal{T}(\mathcal{N} \cup \mathcal{X})$ we write (**if** $u = v$ **then** P **else** Q) for (**let** _ = **eq**(u, v) **in** P **else** Q), assuming that **eq** is a destructor defined by the single rule **eq**$(x, x) \to x$.

Handling binders. We write 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 **let** construct. Similarly, we write fn(P) for the set of *free names* of P, i.e. the set of names that are not bound by a **new** construct. A process P is *closed* if fv$(P) = \emptyset$ and we will only consider the execution of processes under this condition: this means that when **out**(c, u) is emitted, fv$(u) = \emptyset$, hence u is a message; similarly, when executing **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 **in**, **let** and **new** being binders, they induce a notion of α-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 α-renaming): this means, for instance, that

$$\begin{aligned} & \big(\textbf{new}\ n.\ \textbf{out}(c, \textbf{senc}(x, n)).P\big)\{x \mapsto n\} \\ = \;& \big(\textbf{new}\ m.\ \textbf{out}(c, \textbf{senc}(x, m)).P\{n \mapsto m\}\big)\{x \mapsto n\} \\ = \;& \big(\textbf{new}\ m.\ \textbf{out}(c, \textbf{senc}(n, m)).P\{n \mapsto m\}\{x \mapsto n\}\big). \end{aligned}$$

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:*

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

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 $\Sigma_d = \emptyset$:*

$$\begin{array}{ll} \textbf{if}\ u = v\ \textbf{then}\ P\ \textbf{else}\ Q \mid R \rightsquigarrow P \mid R & \textit{when}\ u =_{\mathsf{E}} v \\ \textbf{if}\ u = v\ \textbf{then}\ P\ \textbf{else}\ Q \mid R \rightsquigarrow Q \mid R & \textit{when}\ u \neq_{\mathsf{E}} v \end{array}$$

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 4) due to the presence of **else** branches.

Proposition 1. *If $P \rightsquigarrow Q$ then $P\sigma \rightsquigarrow Q\sigma$ for any bijective renaming σ.*

This result will often be used in the case where there is an “undesirable” name $n \in \mathsf{fn}(P) \setminus \mathsf{fn}(Q)$. Then we can swap m with any other name $n \notin \mathsf{fn}(P) \cup \mathsf{fn}(Q)$, using prop. 1 with the permutation $\{n \leftrightarrow m\}$, which gives us

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

Example 7. *Using the non-determinitic destructor of ex. 5, we can encode non-deterministic choice in processes*[1]*:*

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

We then have $P + Q \rightsquigarrow^2 P$ and $P + Q \rightsquigarrow^2 Q$. It is tempting to consider the more direct encoding $\textbf{if}\ \textbf{choose}(\textbf{true}, \textbf{false}) = \textbf{true}\ \textbf{then}\ P\ \textbf{else}\ Q$ *but this process can only reduce to P since it is not the case that* $\textbf{eq}(\textbf{choose}(\textbf{true}, \textbf{false}), \textbf{true}) \not\Downarrow$.

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:*

A is the adversary, or attacker. By interacting in a malicious way with P, it attempts to obtain s and emit it on c.

[1] 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.

- *terms in A belong to $\mathcal{T}_{\mathsf{pub}}(\mathcal{N})$ and $\mathsf{fn}(P, s) \cap \mathsf{fn}(A) = \emptyset$,*

- *$s =_{\mathsf{E}} s'$ and*

- *$P \mid A \ \rightsquigarrow^* \ \mathbf{out}(c, s') \mid Q$ and names chosen in this reduction when reducing* **new** *constructs are never taken in the initial "secret set" $\mathsf{fn}(P, s)$.*

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:*

- $P_1 = \mathbf{new}\ k.\mathbf{out}(c, \mathbf{senc}(n, k)).\mathbf{out}(c, k)$

- $P_2 = \mathbf{in}(c, x).\mathbf{out}(c, \mathbf{senc}(n, x))$

- $P_3 = \mathbf{out}(c, \mathbf{senc}(n, k)).\mathbf{in}(c, x).\mathbf{if}\ x = n\ \mathbf{then}\ \mathbf{out}(c, k)$

- $P_4 = \mathbf{in}(c, x).\mathbf{let}\ y = \mathbf{adec}(x, k)\ \mathbf{in}\ \mathbf{out}(c, k)\ \mathbf{else}\ \mathbf{out}(c, \mathbf{aenc}(n, \mathbf{pub}(k)))$

- $P_5 = !P_4$

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 $\mathsf{fn}(s) \cap \mathsf{fn}(A) = \emptyset$, i.e. show that the definition would not adequately model secrecy if $\mathsf{fn}(P)$ were used instead of $\mathsf{fn}(P, s)$ 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 $\mathcal{W}$ 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 $\mathcal{T}_{\mathsf{pub}}(\mathcal{W} \cup \mathcal{N})$ and will be denoted by R, M, N.

Definition 3 (Frame)**.** *A* frame *$\vec{n}.\sigma$ is given by a list of names $\vec{n}$ and a finite mapping $\sigma : \mathcal{W} \to \mathcal{T}_c(\mathcal{N})$. Frames are denoted by Φ or Ψ. If $\Phi = \vec{n}.\sigma$ is a frame we write* $\mathsf{bn}(\Phi)$ *for $\vec{n}$;* $\mathsf{dom}(\Phi)$ *for* $\mathsf{dom}(\sigma)$*; $\Phi \cup \{w \mapsto u\}$ for $\vec{n}.(\sigma \cup \{w \mapsto u\})$; and $m.\Phi$ for $(m, \vec{n}).\sigma$.*

Definition 4 (Configuration)**.** *A* configuration *is a pair (P, Φ) where P is a closed process and Φ 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 $x \sharp y$ when no name occurs free in both x and y, i.e. $\mathsf{fn}(x) \cap \mathsf{fn}(y) = \emptyset$. For instance, when R is a recipe, $R \sharp (P, \Phi)$ means that $R \in \mathcal{T}_{\mathsf{pub}}(\mathcal{W} \cup \mathcal{N} \setminus \mathsf{fn}(P, \Phi))$.

Definition 5 (Labelled transitions)**.** *The labelled transition relation $K \xrightarrow{\alpha} K'$, given by the rules of fig. 1, is a relation between two configurations and an* action *α that may be either*

In an input, the recipe explains how the environment computes the input message from the current frame. In an output, w is the handle to which the output message will be associated in the updated frame.

$$(\mathbf{out}(c,u).P \mid Q, \Phi) \xrightarrow{\mathbf{out}(c,w)} (P \mid Q, \Phi \cup \{w \mapsto u\}) \quad \text{when } w \notin \mathsf{dom}(\Phi)$$

$$(\mathbf{in}(c,x).P \mid Q, \Phi) \xrightarrow{\mathbf{in}(c,R)} (P\{x \mapsto u\} \mid Q, \Phi)$$
$$\text{when } R \in \mathcal{T}_{\mathsf{pub}}(\mathcal{N} \cup \mathsf{dom}(\Phi)),\ R \sharp \mathsf{bn}(\Phi) \text{ and } R\Phi \Downarrow u$$

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

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

$$(\mathbf{new}\ n.P \mid Q, \Phi) \xrightarrow{\tau} (P\{n \mapsto m\} \mid Q, m.\Phi) \text{ if } m \sharp (\mathbf{new}\ n.P, Q, \Phi, \mathsf{bn}(\Phi))$$

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

Figure 1: Labelled transitions between configurations

- *the silent action τ, or*
- *the input action* **in**(c, R) *for some $c \in \mathcal{C}$ and $R \in \mathcal{T}_{\mathsf{pub}}(\mathcal{N} \cup \mathcal{W})$, or*
- *the output action* **out**(c, w) *for some $c \in \mathcal{C}$ and $w \in \mathcal{W}$.*

We define the labelled reflexive transitive closure of $\xrightarrow{\alpha}$ as follows: $K_0 \xrightarrow{\mathsf{tr}} K_n$ when $\mathsf{tr} = \alpha_1 \ldots \alpha_n$ *and $K_i \xrightarrow{\alpha_i} K_{i+1}$ for all $i \in [1;n]$.*

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 Φ. 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 $K \xrightarrow{\alpha} K'$ and σ is a bijection on names, then $K\sigma \xrightarrow{\alpha\sigma} K'\sigma$.*

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 Φ allows to* deduce *a message s, written $\Phi \vdash s$, if there exists a recipe R such that $R \sharp$* $\mathsf{bn}(\Phi)$ *and $R\Phi \Downarrow s$.*

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

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

We proceed by induction on tr, essentially translating (tr, R) into an attacker A. If $\mathsf{tr} = \epsilon$ we have $R \sharp \mathsf{bn}(\Phi)$ and $R\Phi \Downarrow s$. We conclude with $A := \mathbf{let}\ x = R\ \mathbf{in}\ \mathbf{out}(c, x)$ since $A\Phi \rightsquigarrow \mathbf{out}(c, s)$.

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

- If $\alpha = \mathbf{out}(c, w)$, P is of the form $\mathbf{out}(c, v).Q \mid R$, $P_1 = Q \mid R$ and $\Phi_1 = \Phi \cup \{w \mapsto v\}$ for some $w \notin \mathsf{dom}(\Phi)$. By induction hypothesis on tr' we have an adversary A_1 against P_1 such that $A_1 \sharp \mathsf{bn}(\Phi_1)$. We conclude with $A := \mathbf{in}(c,w).A_1$: we check that $A \sharp \mathsf{bn}(\Phi) = \mathsf{bn}(\Phi_1)$; that terms of A are in $\mathcal{T}_{\mathsf{pub}}(\mathcal{N} \cup \mathsf{dom}(\Phi))$, because $\mathsf{dom}(\Phi) = \mathsf{dom}(\Phi_1) \setminus \{w\}$; 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 $P \mid A\Phi \rightsquigarrow P_1 \mid A_1\Phi\{w \mapsto v\} = P_1 \mid A_1\Phi_1$.

- If $\alpha = \mathbf{in}(c,R)$ we have $R \sharp \mathsf{bn}(\Phi)$, $R\Phi \Downarrow u$ and

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

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

- If the first transition is a name creation, P is of the form $\mathbf{new}\ n.Q \mid R$ and $P_1 = Q\{n \mapsto m\} \mid R$ for some $m \sharp (P, \Phi, \mathsf{bn}(\Phi))$. By induction hypothesis we obtain an adversary A_1 against P_1, satisfying $A_1 \sharp \mathsf{bn}(\Phi_1)$ and hence $A_1 \sharp \mathsf{bn}(\Phi)$ since $\Phi_1 = m.\Phi$. Since $m \notin \mathsf{fn}(A_1)$, we can reduce $P \mid A_1\Phi \rightsquigarrow P_1 \mid A_1\Phi_1$, which allows us to conclude with $A := A_1$.

- The other cases, i.e. **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 $A \sharp P$ 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 Φ, it would seem reasonable to consider adversaries of the form $A\Phi$ where $A \sharp P$ and all terms in A belong to $\mathcal{T}_{\mathsf{pub}}(\mathcal{N} \cup \mathcal{W})$. But this form of adversaries is not stable by reduction of **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 Φ-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. $A \sharp X$ means $R(A) \sharp X$. 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 Φ-adversary A is such that $R(A)$ only contains terms in $\mathcal{T}_{\mathsf{pub}}(\mathcal{N} \cup \mathsf{dom}(\Phi))$.

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

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 $(P,\Phi) \xrightarrow{\mathsf{tr}} (P',\Phi')$ such that $\Phi' \vdash s$ whenever there exists a Φ-adversary A such that $A \sharp \mathsf{bn}(\Phi)$ and $P \mid A \rightsquigarrow^* \mathbf{out}(\_, s') \mid \ldots$ for some $s' =_{\mathsf{E}} s$, using a reduction that does not pick fresh names in $\mathsf{bn}(\Phi)$.

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

- If the reduction sequence is empty then $\mathbf{out}(c,s')$ is an immediate parallel sub-process of $P \mid A$: if it belongs to P we conclude with $\mathsf{tr} = \mathbf{out}(c,w)$ for some fresh w since $\Phi \cup \{w \mapsto s'\} \vdash s$ by taking $R = w$; if it belongs to A we have $R(s') \Downarrow s$, hence $\Phi \vdash s$, so we can conclude with $\mathsf{tr} = \epsilon$ because $R(s') \sharp \mathsf{bn}(\Phi)$ in inherited from $A \sharp \mathsf{bn}(\Phi)$.

  If t occurs in a Φ-adversary and $t \Downarrow u$, then $R(t)\Phi \Downarrow u$.

- 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 tr unchanged. If the reduction is a name creation, we have assumed that the chosen name satisfies $m \notin \mathsf{bn}(\Phi)$, hence $A' \sharp \mathsf{bn}(\Phi)$. If the reduction is the computation of some **let** $x = t$, we also need to check that A' is still a Φ-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

$$(P, \Phi) \xrightarrow{\mathbf{out}(c,w).\mathbf{in}(c,w)} (P', \Phi \cup \{w \mapsto u\})$$

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 tr $= \tau$.tr$'$. If P creates a name m, we need A to be a $(m.\Phi)$-adversary satisfying $A \sharp \mathsf{bn}(m.\Phi)$: the former is immediate, the latter comes from the side condition on the reduction of $P \mid A$.

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

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

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

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

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

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

Exercise 7. *For each process P of exercise 5, exhibit (if it exists) an execution $(P, \emptyset) \xrightarrow{\mathsf{tr}} (Q, \Phi)$ and a recipe $R \sharp \mathsf{bn}(\Phi)$ 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 (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 $\Sigma_c = \{\mathbf{pair}, \mathbf{aenc}\}$ and $\Sigma_d = \{\mathbf{proj}_1, \mathbf{proj}_2, \mathbf{adec}\}$, no equation, and the following reduction rules:

$$\mathbf{proj}_i(\mathbf{pair}(x_1, x_2)) \to x_i \qquad \mathbf{adec}(\mathbf{aenc}(x, \mathsf{pub}(y)), y) \to x$$

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 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 for the primitives considered here. The first rule is called the *axiom* rule, then there is, for each constructor $f \in \Sigma_c = \{\mathbf{aenc}, \mathbf{pair}\}$, 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.

$$\frac{\Phi(w) = u}{\Phi \vdash w : u} \qquad \frac{n \in \mathcal{N} \setminus \mathsf{bn}(\Phi)}{\Phi \vdash n : n} \qquad \frac{\Phi \vdash R : u}{\Phi \vdash \mathsf{pub}(R) : \mathsf{pub}(u)}$$

$$\frac{\Phi \vdash R_1 : u_1 \quad \Phi \vdash R_2 : u_2}{\Phi \vdash \mathbf{pair}(R_1, R_2) : \mathbf{pair}(u_1, u_2)} \qquad \frac{\Phi \vdash R : \mathbf{pair}(u_1, u_2)}{\Phi \vdash \mathbf{proj}_i(R) : u_i}$$

$$\frac{\Phi \vdash R_1 : u \quad \Phi \vdash R_2 : v}{\Phi \vdash \mathbf{aenc}(R_1, R_2) : \mathbf{aenc}(u, v)} \qquad \frac{\Phi \vdash R_1 : \mathbf{aenc}(u, \mathsf{pub}(v)) \quad \Phi \vdash R_2 : v}{\Phi \vdash \mathbf{adec}(R_1, R_2) : u}$$

Figure 2: Decorated deduction rules.

Proposition 4. *Let Φ be a frame. We have $\Phi \vdash u$ (in the sense of def. 6) iff $\Phi \vdash R : u$ is derivable (in the sense of fig. 2).*

When one is concerned only with the deducibility of u from Φ, 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:

$$\frac{u \in T}{\Phi \vdash u} \qquad \frac{n \in \mathcal{N} \setminus \mathsf{bn}(\Phi)}{\Phi \vdash n}$$

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 $\Phi \vdash u$ given Φ 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 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):

$$\frac{\dfrac{T \vdash v_1 \quad T \vdash v_2}{T \vdash \mathbf{pair}(v_1, v_2)}}{T \ \vdash v_i} \qquad \frac{\dfrac{T \vdash v \quad T \vdash \mathbf{pub}(w)}{T \vdash \mathbf{aenc}(v, \mathbf{pub}(w))} \quad T \vdash w}{T \vdash v}$$

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 $\Phi \vdash v$ 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 $\Phi \vdash v$ that: if Π ends with an axiom or decomposition then all of its internal judgments are of the form $\Phi \vdash u$ with u a subterm of Φ; otherwise (when Π ends with a composition rule) its internal judgments are of the form $\Phi \vdash u$ 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. 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*.

$$\frac{}{y \Downarrow^{x=y} x} \qquad \frac{}{n \Downarrow^{x=n} x}$$

$$\frac{t_1 \Downarrow^{\varphi_1} x_1 \quad t_2 \Downarrow^{\varphi_2} x_2}{f(t_1, t_2) \Downarrow^{\varphi_1 \wedge \varphi_2 \wedge x = f(x_1, x_2)} x} \; f \in \{\mathbf{pair}, \mathbf{aenc}\}$$

$$\frac{t \Downarrow y}{\mathbf{proj}_i(t) \Downarrow^{\varphi \wedge y = \mathbf{pair}(x_1, x_2) \wedge x = x_i} x}$$

$$\frac{t_1 \Downarrow^{\varphi_1} x_1 \quad t_2 \Downarrow^{\varphi_2} x_2}{\mathbf{adec}(t_1, t_2) \Downarrow^{\varphi_1 \wedge \varphi_2 \wedge x_1 = \mathbf{aenc}(x, \mathsf{pub}(x_2))} x}$$

Figure 3: Symbolic computation success rules

$$\frac{t_1 \Downarrow^{\varphi} \bot}{f(t_1, t_2) \Downarrow^{\varphi} \bot} \; f \in \{\mathbf{pair}, \mathbf{aenc}\} \qquad \frac{t_2 \Downarrow^{\varphi} \bot}{f(t_1, t_2) \Downarrow^{\varphi} \bot} \; f \in \{\mathbf{pair}, \mathbf{aenc}\}$$

$$\frac{}{\mathbf{proj}_i(t) \Downarrow^{t = \mathbf{aenc}(x_1, x_2)} \bot} \qquad \frac{}{\mathbf{proj}_i(t) \Downarrow^{t = n} \bot}$$

$$\frac{}{\mathbf{adec}(t, k) \Downarrow^{t = \mathbf{pair}(x_1, x_2)} \bot} \qquad \frac{}{\mathbf{adec}(t, k) \Downarrow^{t = n} \bot}$$

Figure 4: Symbolic computation failure rules

In order to symbolically describe (un)successful computations, we define in figs. 3 and 4 the relations $t \Downarrow^{\varphi} x$ and $t \Downarrow^{\varphi} \bot$ where $t \in \mathcal{T}(\mathcal{N} \cup \mathcal{X})$, $x \in \mathcal{X}$ and φ is a conjunction of equalities between terms of $\mathcal{T}_c(\mathcal{N} \cup \mathcal{X})$. Intuitively, $t \Downarrow^{\varphi} x$ means that under the condition φ, the computation t will succeed and result in x, whose value will be constrained by φ. Similarly, $t \Downarrow^{\varphi} \bot$ means that under φ the computation t will fail. The rules of figs. 3 and 4 are meant to be sound but also complete with respect to these intuitions.

In order to express formally these soundness and completeness statements, we define $\mathsf{Sol}(\varphi)$ as the set of all $\theta : \mathsf{fv}(\varphi) \to \mathcal{T}_c(\mathcal{N})$ such that $\varphi\theta$ is a conjunction of identities; we also write $\theta \sqsubseteq \theta'$ when $\mathsf{dom}(\theta) \subseteq \mathsf{dom}(\theta')$ and $\theta'|_{\mathsf{dom}(\theta)} = \theta$.

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

- *For every $t \Downarrow^{\varphi} x$ and $\theta \in \mathsf{Sol}(\varphi)$, $t\theta \Downarrow \theta(x)$.*
- *For every $\theta : \mathsf{fv}(t) \to \mathcal{T}_c(\mathcal{N})$ and u such that $t\theta \Downarrow u$, there exists φ such that $t \Downarrow^{\varphi} x$, $\theta \sqsubseteq \theta' \in \mathsf{Sol}(\varphi)$ and $u = \theta'(x)$.*

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

- *For every $t \Downarrow^{\varphi} \bot$ and $\theta \in \mathsf{Sol}(\varphi)$, $t\theta \not\Downarrow$.*
- *For every $\theta : \mathsf{fv}(t) \to \mathcal{T}_c(\mathcal{N})$ such that $t\theta \not\Downarrow$, there exists φ such that $t \Downarrow^{\varphi} x$ and $\theta \sqsubseteq \theta' \in \mathsf{Sol}(\varphi)$.*

Example 8. *We can derive $\mathbf{adec}(\mathbf{proj}_1(x), y) \Downarrow^{\varphi} z$ with*

$$\varphi := x = x \wedge x = \mathbf{pair}(x_1, x_2) \wedge x_1 = x_1 \wedge y = y \wedge x_1 = \mathbf{aenc}(z, \mathsf{pub}(y)).$$

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

Example 9. *We can derive $\mathbf{proj}_1(\mathbf{pair}(x_1, \mathbf{proj}_1(x_2))) \Downarrow^{\varphi} y$ where φ is*

$$x_2 = \mathbf{pair}(x_2', x_2'') \wedge x' = \mathbf{pair}(x_1, x_2') \wedge x' = \mathbf{pair}(y, y')$$

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

$$x_2 = \mathbf{pair}(x_2', x_2'') \wedge x_1 = y.$$

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

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 is useful in the completeness arguments. For instance, when building a derivation witnessing the fact that $t\theta \Downarrow u$ in the case where $t = \mathbf{pair}(t_1, t_2)$, it suffices to choose fresh variables x_1 and x_2, obtain by induction hypotheses $t_1 \Downarrow^{\varphi_1} x_1$ and $t_2 \Downarrow^{\varphi_2} x_2$ as well as θ_1' and θ_2', combine the two derivations to obtain a derivation of $t \Downarrow^{\varphi} x$ for a new fresh variable x with $\varphi := \varphi_1 \wedge \varphi_2 \wedge x = \mathbf{pair}(x_1, x_2)$, and conclude $\theta \sqsubseteq \theta' \in \mathsf{Sol}(\varphi)$ with $\theta' := \theta_1' \cup \theta_2' \cup \{x \mapsto \mathbf{pair}(\theta_1'(x_1), \theta_2'(x_2))\}$. This last step requires that θ_1' and θ_2' agree on their common domain, which we obtain by assuming wlog. that $\mathsf{fv}(\varphi_1) \setminus \mathsf{fv}(\varphi) \sharp \mathsf{fv}(\varphi_2) \setminus (\varphi)$, i.e. freshly introduced variables of each sub-branch are disjoint.

In practice, when trying to enumerate, for a given t, a complete set of $t \Downarrow^{\varphi} x$, it is sufficient to look for derivations that correspond to the (fresh) choices of variables made in the completeness argument. And, in fact, there is only one such derivation. For instance, for $t := \mathbf{pair}(x, y)$ it is sufficient to produce $t \Downarrow^{\varphi} x'$ with $\varphi := x = x_1 \wedge y = x_2 \wedge x' = \mathbf{pair}(x_1, x_2)$.

In the case of computation failures $t \Downarrow^{\varphi} \bot$, there is also a choice of name n in two destructor rules: for example, $\mathbf{proj}_1(x) \Downarrow^{x=n} \bot$, $\mathbf{proj}_1(x) \Downarrow^{x=m} \bot$, etc.; similarly, $\mathbf{proj}_1(n) \Downarrow^{n=n} \bot$, $\mathbf{proj}_1(n) \Downarrow^{n=m} \bot$, etc. This infinite choice can also be avoided in practice, but we won't detail how.

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 $T \vdash^? u$ 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 $\bot$ and the conjunction symbol is used to denote the union of constraint systems. We assume two conditions on a deducibility constraint system $T_1 \vdash^? u_1 \wedge \ldots \wedge T_n \vdash^? u_n$:*

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

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

$$
\begin{array}{lr}
K = \big(\mathbf{in}(c,x).P \mid Q, \Phi, C\big) \xrightarrow{\mathbf{in}(c,x)} \big(P \mid Q, \Phi, C \wedge \Phi \vdash^? x\big) & x \notin \mathsf{fv}(K) \\
\big(\mathbf{out}(c,u).P \mid Q, \Phi, C\big) \xrightarrow{\mathbf{out}(c,w)} \big(P \mid Q, \Phi \cup \{w \mapsto u\}, C\big) & w \notin \mathsf{dom}(\Phi) \\
\big(\mathbf{let}\ x = t\ \mathbf{in}\ P\ \mathbf{else}\ Q \mid R, \Phi, C\big) \xrightarrow{\tau} \big(P \mid R, \Phi, C\big)\theta & t \Downarrow^{\varphi} x,\ \theta \in \mathsf{Sol}(\varphi) \\
\big(\mathbf{let}\ x = t\ \mathbf{in}\ P\ \mathbf{else}\ Q \mid R, \Phi, C\big) \xrightarrow{\tau} \big(Q \mid R, \Phi, C\big)\theta & t \Downarrow^{\varphi} \bot,\ \theta \in \mathsf{Sol}(\varphi) \\
K = \big(\mathbf{new}\ n.P \mid Q, \Phi, C\big) \xrightarrow{\tau} \big(P \mid Q, n.\Phi, C\big) & n \sharp K \\
\big(!P \mid Q, \Phi, C\big) \xrightarrow{\tau} \big(P \mid !P \mid Q, \Phi, C\big) &
\end{array}
$$

It actually suffices to take a most general unifier θ of φ. There is also no need to consider all possible φ, assuming variables are chosen sufficiently fresh.

Figure 5: Symbolic labelled transitions.

Definition 9. *A symbolic configuration is a tuple (P, Φ, C) 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.*

When K is a symbolic configuration (P, Φ, C), we simply write $\mathsf{Sol}(K)$ for $\mathsf{Sol}(C)$. Given a concrete configuration $K = (P, \Phi)$, we define $\lceil K \rceil$ as the symbolic configuration $(P, \Phi, \top)$. Conversely, if $K' = (P, \Phi, C)$ is a symbolic configuration and $\theta \in \mathsf{Sol}(K')$, we define $\lfloor K' \rfloor \theta$ as the concrete configuration $(P\theta, \Phi\theta)$.

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

- *If $\lceil K_1 \rceil \xrightarrow{\mathsf{tr}'} K'_2$ and $\theta \in \mathsf{Sol}(K'_2)$, then $K_1 \xrightarrow{\mathsf{tr}} \lfloor K'_2 \rfloor \theta$ for some* tr*.*
- *If $K_1 \xrightarrow{\mathsf{tr}} K_2$ then there exists K'_2 and $\theta \in \mathsf{Sol}(K'_2)$ such that $\lceil K_1 \rceil \xrightarrow{\mathsf{tr}'} K'_2$ and $K_2 = \lfloor K'_2 \rfloor \theta$.*

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

Example 10. *Consider the process $P := \mathbf{in}(c,x).\mathbf{let}\ y = \mathbf{adec}(x,k)\ \mathbf{in}\ \mathbf{out}(c,y)$ and the initial knowledge $\Phi := \{w \mapsto \mathbf{pub}(k)\}$. Then we have*

$$
\begin{array}{rcl}
(P, \Phi, \top) & \xrightarrow{\mathbf{in}(c,x)} & (P', \Phi, \Phi \vdash^? x) \\
 & \xrightarrow{\tau} & (\mathbf{out}(c,y), \Phi, \Phi \vdash^? \mathbf{aenc}(y, \mathbf{pub}(k))) \\
 & \xrightarrow{\mathbf{out}(c,w')} & (0, \Phi \cup \{w' \mapsto y\}, \Phi \vdash^? \mathbf{aenc}(y, \mathbf{pub}(k))).
\end{array}
$$

2.3 Solving deducibility constraint systems

We refer the reader to chapter 3 of the old lecture notes[2] 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.

[2] `http://www.lsv.fr/~baelde/secu/poly.pdf`

- 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 $\mathsf{img}(\Phi)$.

- Instead of considering a reduction $\mathbf{adec}(\mathbf{aenc}(x, \mathsf{pub}(y)), y) \rightarrow x$, the lectures notes work with $\mathbf{adec}(\mathbf{aenc}(x, y), \mathsf{sk}(y)) \rightarrow x$ where 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**.

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.

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 expresses 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 Φ and Ψ are statically equivalent, written $\Phi \sim \Psi$ whenever all of the following conditions hold:*

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

We consider recipes that correspond to computations performed by the attacker, who does not have access to the secret/fresh names of either frame.

We may abbreviate the second condition as "$R\Phi\Downarrow$ iff $R\Psi\Downarrow$" and the third as "$M\Phi\Downarrow =_{\mathsf{E}} N\Phi\Downarrow$ iff $M\Psi\Downarrow =_{\mathsf{E}} N\Psi\Downarrow$" — recall that static equivalence is symmetric.

Example 11. *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 12. *Assuming that asymmetric encryption is modelled using the reduction $\mathbf{adec}(\mathbf{aenc}(x, \mathbf{pub}(y)), y) \to x$ alone, we have:*

- $(k, n).\{w \mapsto \mathbf{aenc}(m, \mathbf{pub}(k))\} \sim (k, n).\{w \mapsto n\}$*;*
- $(k, n).\{w_0 \mapsto \mathbf{pub}(k), w \mapsto \mathbf{aenc}(m, \mathbf{pub}(k))\} \not\sim$
  $(k, n).\{w_0 \mapsto \mathbf{pub}(k), w \mapsto n\}$*;*
- $(k, n, m).\{w \mapsto \mathbf{aenc}(m, \mathbf{pub}(k)), w' \mapsto k\} \not\sim (k, n, m).\{w \mapsto n, w' \mapsto k\}$*;*
- $(k, n, m).\{w \mapsto \mathbf{aenc}(m, \mathbf{pub}(k)), w' \mapsto \mathbf{aenc}(n, \mathbf{pub}(k)), w_0 \mapsto \mathbf{pub}(k)\} \not\sim$
  $(k, n, m).\{w \mapsto \mathbf{aenc}(m, \mathbf{pub}(k)), w' \mapsto \mathbf{aenc}(m, \mathbf{pub}(k)), w_0 \mapsto \mathbf{pub}(k)\}$.

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

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* $\mathbb{T}$ *may occur. A process P* may pass *a test T, written $P \models T$ if*

$$P \mid T \leadsto^* \mathbf{out}(\mathbb{T}, u) \mid Q \ \textit{ for some } u \textit{ and } Q.$$

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

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* $\mathsf{T}(P) = \mathsf{T}(Q)$.

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

Proposition 9. *If* $(P, \emptyset) \xrightarrow{\mathsf{tr}} (P', \Phi)$ *and* $(Q, \emptyset) \xrightarrow{\mathsf{tr}} (Q', \Psi)$ *with* $\Phi \not\sim \Psi$*, 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 $\mathbf{in}(c, x).\mathbf{out}(\mathbb{T}, \mathbf{ok})$ belongs to $\mathsf{T}(P) \setminus \mathsf{T}(Q)$.

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 continuation, thus a process can observe when its output has been received; such observables 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 $P := \mathbf{out}(c, \mathbf{ok}).\mathbf{out}(c, \mathbf{ok})$ and $Q := \mathbf{out}(c, \mathbf{ok}) + \mathbf{out}(c, \mathbf{ok}).\mathbf{out}(c, \mathbf{ok})$ (using non-deterministic choice as encoded e.g. in ex. 7). Both processes may pass the test $\mathbf{in}(c, x).\mathbf{in}(c, y).\mathbf{out}(\mathbb{T}, \mathbf{ok})$. 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. This would be captured through must testing, but we won't use or detail this notion, that has not been used much in the security litterature.

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 statically 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 τ actions, which are not observable) and resulting in a frame that is statically equivalent.

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

Trace equivalence is then often defined from $\mathsf{Tr}(K) = \{(\mathsf{tr}, \Phi(K')) \mid K \overset{\mathsf{tr}}{\Rightarrow} K'\}$ as the relation $K \approx K'$ which holds when for all $(\mathsf{tr}, \Phi) \in \mathsf{Tr}(K)$ there exists $(\mathsf{tr}, \Psi) \in \mathsf{Tr}(K')$ 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*

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

We say that K and K' are trace equivalent when $\mathsf{Tr}'(K) = \mathsf{Tr}'(K')$.

Note that this definition only makes sense when $\mathsf{dom}(\Phi(K)) = \mathsf{dom}(\Phi(K'))$. More surprisingly, it essentially requires $\mathsf{bn}(\Phi(K)) = \mathsf{bn}(\Phi(K'))$, otherwise trace equivalence breaks for silly reasons. For instance, supposing that there exists a name $n \in \mathsf{bn}(\Phi(K')) \setminus \mathsf{bn}(\Phi(K))$, 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 α-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 $u =_{\mathsf{E}} v$ are equivalent. Then trace equivalence implies may-testing equivalence.*

We first show that the determinism assumption is necessary.

Example 13. *Consider $\Sigma = \{\mathbf{f}, \mathbf{g}, \mathbf{ch}, \mathbf{val}\}$ with $\Sigma_{\mathsf{pub}} = \{\mathbf{ch}, \mathbf{val}\}$. The unary symbols $\mathbf{f}$ and $\mathbf{g}$ are private constructors, and the equational theory E is empty. The public constructors $\mathbf{ch}$ (unary) and $\mathbf{val}$ (binary) are associated with the following reductions:*

$$\begin{array}{rclcrcl} \mathbf{ch}(x) & \to & \mathbf{f}(x) & \qquad & \mathbf{val}(x, \mathbf{f}(x)) & \to & x \\ \mathbf{ch}(x) & \to & \mathbf{g}(x) & & \mathbf{val}(x, \mathbf{g}(x)) & \to & x \end{array}$$

Consider now two processes:

$$\begin{array}{rcl} P & := & \mathbf{new}\ n.\ \mathbf{out}(c, n).\mathbf{in}(c, x).\mathbf{in}(c, y). \\ & & \mathbf{let}\ \_ = \mathbf{val}(n, x)\ \mathbf{in}\ \mathbf{let}\ \_ = \mathbf{val}(n, y)\ \mathbf{in}\ \mathbf{if}\ x = y\ \mathbf{then}\ \mathbf{out}(c, n)\ \mathbf{else}\ 0 \\ Q & := & \mathbf{new}\ n.\ \mathbf{out}(c, n).\mathbf{in}(c, x).\mathbf{in}(c, y). \\ & & \mathbf{let}\ \_ = \mathbf{val}(n, x)\ \mathbf{in}\ \mathbf{let}\ \_ = \mathbf{val}(n, y)\ \mathbf{in}\ \mathbf{if}\ x = y\ \mathbf{then}\ 0\ \mathbf{else}\ \mathbf{out}(c, n) \end{array}$$

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

$$\mathbf{in}(c, z).\mathbf{let}\ z' = \mathbf{ch}(z)\ \mathbf{in}\ \mathbf{out}(c, z').\mathbf{out}(c, z').\mathbf{in}(c, \_).\mathbf{out}(\mathbb{T}, \_).0$$

Let us now show that $\mathsf{Tr}'(P) = \mathsf{Tr}'(Q)$. 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 $\mathbf{out}(c, w).\mathbf{in}(c, R).\mathbf{in}(c, R').\tau$ (for arbitrary w, R and R') together with any frame $\Phi \sim n.\{w \mapsto n\}$. In order to execute one more τ action, we must have $R\Phi \Downarrow \mathbf{f}(n)$ or $R\Phi \Downarrow \mathbf{g}(n)$: since $\mathbf{f}$ and $\mathbf{g}$ are private, we must have $R = \mathbf{ch}(w)$, regardless of whether we are considering P or Q. In order to execute a third τ action, we must similarly have $R' = \mathbf{ch}(w)$. Hence $\mathsf{Tr}'(P)$ and $\mathsf{Tr}'(Q)$ coincide for traces of length up to six.

The only trace of length seven that could be considered is

$$\mathbf{out}(c, w).\mathbf{in}(c, \mathbf{ch}(w)).\mathbf{in}(c, \mathbf{ch}(w)).\tau.\tau.\tau.\mathbf{out}(c, w')$$

(there 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 $\Phi \sim n.\{w \mapsto n, w' \mapsto n\}$: crucially, for Q, this is because it is possible to take $\mathbf{ch}(w)\Phi \Downarrow \mathbf{f}(n)$ for the first input $\mathbf{ch}(w)\Phi \Downarrow \mathbf{g}(n)$ for the second.

Proof of prop. 10. Under the computation determinism assumption we show that, when $P_0 \approx Q_0$, any test $T_0 \in \mathsf{T}(P)$ also belongs to $\mathsf{T}(Q)$. From $P_0 \mid T_0 \rightsquigarrow^* \mathbf{out}(\mathbb{T}, \_) \mid \_$ we obtain a reduction

$$P_0 \mid T_0\theta_0 \rightsquigarrow^* P_0' \mid T_0'\theta_0' \rightsquigarrow P_1 \mid T_1\theta_1 \rightsquigarrow^* P_1' \mid T_1'\theta_1' \ldots \rightsquigarrow P_n \mid T_n\theta_n \rightsquigarrow^* P_n' \mid T_n'\theta_n'$$

such that:

1. $(P, \emptyset) = (P_0, \Phi_0) \overset{\alpha_1}{\Rightarrow} (P_1, \Phi_1) \ldots \overset{\alpha_n}{\Rightarrow} (P_n, \Phi_n)$
2. for every $x \in \mathsf{dom}(\theta_n')$ there exists R_x such that $R_x\Phi_n \Downarrow \theta_n'(x)$
3. for every i, $T_i\theta_i \rightsquigarrow^* T_i'\theta_i'$ and $\theta_i \sqsubseteq \theta_i' \sqsubseteq \theta_{i+1}$
4. for every i, $(T_i, T_i') \sharp \mathsf{bn}(\Phi_i)$
5. $T_n' = \mathbf{out}(\mathbb{T}, \_) \mid \_$
6. for every i there exists a conjunction of equalities φ_i such that
   for all σ, $\models \varphi_i\sigma$ iff $T_i\sigma \rightsquigarrow^* T_i'\sigma$

Here, $\models \varphi$ means that φ consists only of identities modulo the equational theory. Note that condition 2 is expressed only for θ_n' but actually applies to all θ_i and θ_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 $T_i\theta_i \rightsquigarrow^* T_i'\theta_i$. There we ensure wlog that fresh names are picked outside of $\mathsf{bn}(\Phi_i)$. When **let** constructs are evaluated, we can construct T_i' satisfying 4: the key case is when $T\theta_i =$ **let** $x = t\theta_i$ **in** $P\theta_i$ **else** $Q\theta_i$ reduces to $P\theta_i\{x \mapsto u\}$, for which we set $T' = P$ and extend θ_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 φ_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 $t \Downarrow^{\varphi} x$ and $t \Downarrow^{\varphi} \bot$ predicates of figs. 3 and 4.)
- The internal reductions $P_i \rightsquigarrow^* P_i'$ are mapped to τ transitions performed implicitly before the α_{i+1} action of $(P_i, \Phi_i) \overset{\alpha_{i+1}}{\Rightarrow} (P_{i+1}, \Phi_{i+1})$.
- When the communication is an output from P_i, we have $\alpha_{i+1} = \mathbf{out}(c, u)$ for some c and u. We have $\Phi_{i+1} = \Phi_i \cup \{w \mapsto u\}$ for some new handle w. The process T_i' is of the form $\mathbf{in}(c, x).Q \mid R$ and we set $T_{i+1}' = Q \mid R$ 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, $T_i' = \mathbf{out}(c, u).Q \mid R$ and the recipe $R_u = u\{x \mapsto R_x\}_{x \in \mathsf{fv}(u)}$ is such that $R_u\Phi_i \Downarrow u$ by condition 2; we thus take $\alpha_{i+1} = \mathbf{in}(c, R_u)$.

By $P \approx Q$ we have $(\alpha_1 \ldots \alpha_n, \Psi_n) \in \mathsf{Tr}(Q)$ for some frame Ψ_n such that $\Phi_n \sim \Psi_n$. Moreover we can choose Ψ_n wlog to obtain $(T_i, T_i') \sharp \mathsf{bn}(\Psi_n)$ for all i. Consider now $\varphi := \bigwedge_i \varphi_i \wedge \varphi_i'$. We have $\models \varphi\theta_n'$, hence by static equivalence $\models \varphi\sigma_n'$ for some σ_n' such that, for every $x \in \mathsf{dom}(\sigma_n') = \mathsf{dom}(\theta_n')$, $R_x\Psi_n \Downarrow \sigma_n'(x)$. From σ_n' we derive, for all i, $\sigma_i = \sigma_n'|_{\mathsf{dom}(\theta_i)}$ and $\sigma_i' = \sigma_n'|_{\mathsf{dom}(\theta_i')}$. By condition 6 we obtain that, for all i, $T_i\sigma_i' \rightsquigarrow^* T_i'\sigma_i'$. The execution $(Q, \emptyset) = (Q_0, \Psi_0) \overset{\alpha_1 \ldots \alpha_n}{\Rightarrow} (Q_n, \Psi_n)$ yields $Q_i \rightsquigarrow^* Q_i'$ for all i, with $(Q_i', \Psi_i) \xrightarrow{\alpha_{i+1}} (Q_{i+1}, \Psi_{i+1})$. We finally check that $Q_i' \mid T_i'\sigma_i' \rightsquigarrow Q_{i+1} \mid T_{i+1}\sigma_{i+1}$:

- If $\alpha_{i+1} = \textbf{in}(c, R_u)$ we have an output $\textbf{out}(c, u)$ at toplevel in T'_i, with $R_u\Phi_i \Downarrow u\theta'_i$. Then $R_u\Psi_i \Downarrow u\sigma'_i$. Since Q'_i perform α_{i+1} in the LTS, we have $Q'_i = \textbf{in}(c, x).Q \mid R$ 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 $u\sigma'_i =_{\mathsf{E}} v$. 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 mimick 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. $P \leadsto^* \textbf{out}(c, u).P' \mid P''$.

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

Observational equivalence *is the largest observational bisimulation.*

Expanding the composition of relations, this means that there exists Q' such that $Q \leadsto^* Q'$ and $Q'\ \mathcal{R}\ P'$.

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

Example 15. *Let us write α for the output prefix $\textbf{out}(c_\alpha, \textbf{ok})$, and similarly for β and γ. 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 $\textbf{in}(c_\alpha, x).0$, the second process can reduce (in two steps) to γ, exhibiting an output on c_γ, 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* $B \xrightarrow{\tau}{}^{*}\mathcal{R}\ A'$*;*
- $A \xrightarrow{\alpha} A'$ *implies* $B \overset{\alpha}{\Rightarrow}\mathcal{R}\ A'$*.*

Bisimilarity *is the largest bisimulation.*

Abadì, Blanchet and Fournet have shown [1] 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.

Proposition 11. *Bisimilarity implies trace equivalence.*

Proof. If $A\ \mathcal{R}\ B$ and $A \overset{\mathsf{tr}}{\Rightarrow} A'$ then $B \overset{\mathsf{tr}}{\Rightarrow} B'$ 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 $A \overset{\mathsf{tr}}{\Rightarrow} A'$, and for any observable action α, $A' \xrightarrow{\alpha} A_1$ and $A' \xrightarrow{\alpha} A_2$ imply $A_1 = A_2$.*

Note that determinacy does not mean that $A \overset{\mathsf{tr}}{\Rightarrow} A_1$ and $A \overset{\mathsf{tr}}{\Rightarrow} A_2$ imply $A_1 = A_2$ — the hidden τ actions can add many (inessential) variations between A_1 and A_2. However it can be shown that $A \xrightarrow{\tau^*.\alpha} A_1$ and $A \xrightarrow{\tau^*.\alpha} A_2$ 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 (and, a fortiori, non-determinstic 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 $A \xrightarrow{\tau}{}^{*} A'$. In that case, one can show[3] that $A' \approx A$. Thus $A' \approx B$ and A' is still determinate, thus we have $A'\ \mathcal{R}\ B'$ as required by choosing $B' := B$.

For the last condition, since we have already seen how to mimick τ actions in the previous paragraph, we can focus on showing that $A \xrightarrow{\alpha} A'$ implies $B \overset{\alpha}{\Rightarrow} B'$ with $A' \approx B'$. By $A \approx B$, there actually exists B' such that $B \xrightarrow{\tau^*.\alpha} B'$. Let us now show that $A' \approx B'$. Assuming $A' \overset{\mathsf{tr}}{\Rightarrow} A''$, we have $A \overset{\alpha.\mathsf{tr}}{\Rightarrow} A''$ and thus $B \xrightarrow{\tau^*.\alpha} B'_1 \overset{\mathsf{tr}}{\Rightarrow} B'' \sim A''$. By the remarks above, since B is determinate, we have $B' \approx B'_1$. Hence we also have $B' \overset{\mathsf{tr}}{\Rightarrow}\sim A''$ 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.

[3] The possible τ actions do not change the frame's contents. Moreover, consider a trace tr of A, we can anticipate some of its τ actions, and add some more, to obtain a trace tr of A' resulting in the same frame.

5 Verifying equivalences

We have briefly seen the notion of diff-equivalence introduced in Proverif, and how it is verified. Although this is out of the scope of the course, more details about this can be found in [3].

We have also covered the general structure of the verification technique used in Akiss, and how the underlying notions differ from trace equivalence. The reference for this part of the lectures is [4].

6 Advanced topics

In the last lecture, we will talk about the approach of [2] and some of its current uses.

References

[1] M. Abadi, B. Blanchet, and C. Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018.

[2] G. Bana and H. Comon-Lundh. A computationally complete symbolic attacker for equivalence properties. In *ACM Conference on Computer and Communications Security*, pages 609–620. ACM, 2014.

[3] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 2008.

[4] R. Chadha, V. Cheval, Ştefan Ciobâcă, and S. Kremer. Automated verification of equivalence properties of cryptographic protocols. *ACM Trans. Comput. Log.*, 17(4):23:1–23:32, 2016.