Security Policy in a Declarative Style

Author R. Echahed
Institut d'Informatique et de Mathématiques
Appliquées de Grenoble,
Laboratoire test Leibniz, 46, av Félix Viallet,
38000 Grenoble, France
rachid.echahed@imag.fr

F. Prost
Institut d'Informatique et de Mathématiques
Appliquées de Grenoble,
Laboratoire Leibniz, 46, av Félix Viallet,
38000 Grenoble, France
frederic.prost@imag.fr

ABSTRACT

We address the problem of controlling information leakage in a concurrent declarative programming setting. Our aim is to define verification tools in order to distinguish between authorized, or declared, information flows such as password testing (e.g., ATM, login processes, etc.) and non-authorized ones. In this paper, we first propose a way to define security policies as confluent and terminating rewrite systems. Such policies define how the privacy levels of information evolve. Then, we provide a formal definition of secure processes with respect to a given security policy. We also define an actual verification algorithm of secure processes based on constraint solving.

1. INTRODUCTION

The problem of the preservation of the confidentiality of data represents nowadays a prominent feature of computer systems. This is especially true in a context where programs and data may move around along communication networks. The usual theoretical approach of this problem, initiated by Goguen and Meseguer in [12], uses the notion of non-interference. Roughly, the idea is that two parts of a system are non-interfering if what it is done by one part has no effect on the other part and vice versa. Hence a program, or a system respects secrecy if there are no interferences, or no information leakage, from secret data to public ones. This approach relies heavily on a precise formalization of what it is intended by the words "no effect". Indeed, it amounts to the problem of equivalence of behaviors which is not a trivial problem in non-deterministic settings [26].

A great deal of work has been done along these lines. There are mainly two directions. The first one is about the programming paradigms involved: imperative e.g., [30], functional e.g., [22], process algebras e.g., [14, 1], functional and communicating processes e.g., [32], imperative and communicating processes e.g., [28, 2] and multi-paradigm systems e.g., [7]. The second one is about the context in which non-interference is studied. Indeed different confinement properties, such as deterministic, nondeterministic and probabilistic e.g., [31], can be defined. See [27] for a complete overview.

A common feature of these works is that they consider information leakage from a very strict point of view despite the fact that in real world applications, such absolute non-interference properties can hardly be obtained e.g., [25]. The notion of approximate non-interference e.g., [20, 19], along with declassification and other weakenings of non-interference e.g., [34, 11] have been only recently investigated. Following this line of study, we propose a formalization in which the user may declare its security policy. The idea is that the user may *declare* that some functions are allowed to provoke information leakage. We call such functions declassifying functions. This formalization extends previous works in the following directions: the expressivness of the programing system and the expressivity of security policies.

A typical example where declassifying functions are used is the mechanism of password. It is widely spread in real life. Think of Automated Teller Machines, log on procedures (on computers, web sites with restricted access...), restricted areas, PIN for cellular phones etc. Schematically, the situation is the following : to access secret areas one has to enter the right password to a controller. From a strict non-interference point of view, there is a forbidden information leakage in such situations. Indeed, it is possible to try all possible passwords to be finally granted the access to the restricted area. Therefore private information, e.g. the knowledge of the password, may influence public behavior, e.g. a hacker randomly submitting a number may guess the password by chance. Nevertheless, this kind of leakage are controlled through the declaration of declassifying functions. Therefore such information leakage has to be considered as harmless. Firstly because information leakage is only allowed for some *known* parts, namely the ones checking the password. Therefore it is not possible to directly disclose private data. Secondly, because it is possible to define a system policy that deters brute-force attacks: for instance the system may become inactive after some unsuccessful attempts (after three non valid PIN, the ATM retains the credit card). Another example of such controlled information leakage is the case of public key cryptography. Suppose that encrypted data are public, since both the encryption key and algorithm are known it is theoretically possible to encrypt every file of some fixed size to find out which one gives the same encrypted file. It is a broader version of the password verification done in UNIX systems in which encrypted password are stored and readable by anyone.

Note that this scheme is not related with cryptographic issues. Declassifying functions simply play the role of an interface between security levels, they are not required to be cryptographic functions or whatsoever. It is out of the scope of our analysis to check if declassifying are really "safe" from a practical point of view. What we want to do is to be able to study the security of a process with respect to a given security policy.

We want to formalize the following idea: if a function is declared as declassifying by a security policy, we make abstraction of the leakage coming from the use of this function. In other words a process is secure if the only information leakage it produces are sound with the declared security policy.

We made a first study of this idea in [5]. In the present paper we enrich our previous work and propose a new way to define a security policy based on rewriting systems. It gives a way to define richer security policies than in earlier works. We also provide an algorithm to check if a process is secure with respect to a given security policy. It is based on an abstract operational semantics generating constraints over privacy levels.

We start by the presentation of our multi-paradigm computation model in section 2. The interest in using such a calculus is that its expressiveness encompasses many programming paradigms including imperative, declarative and concurrent ones. It makes our analysis easy to adapt to many other programming paradigms. Then, in section 3, we give a precise definitions of what we intend by security policy and security properties of processes and their implications in terms of confidentiality. We give an algorithm to check whether a process is secure with respect to a given security policy in section 4. We discuss related works in section 5 and conclude in section 6.

2. A FRAMEWORK FOR CONCURRENT DECLARATIVE PROGRAMMING

The computational model used in this paper is a simplified version of the one proposed in [8, 9]; we refer to these works for a more detailed presentation of the complete model and its implementation. Roughly speaking, a program or a component in our framework consists of two parts $K = (F, \Pi\!R)$. $\Pi\!R$ is a set of process definitions and F is a declarative program, i.e., a set of formulæ (here, we consider Term Rewrite Systems (TRS)), which we call a *store*. We assume the reader is familiar with TRS (see, e.g., [4]).

The execution model of a component can be schematized as follows. Processes communicate by modifying a common store F, i.e., by altering it in a non-monotonic way, for example by simply redefining constants (e.g., adding a message in a queue) or by adding or deleting formulæ in F. Hence, the execution of processes will cause the transformation of the store F. Every change of the store is the result of the execution of an *action*. A store F is used to evaluate expressions (i.e. normalization of terms in our setting).

We now define the different parts of our computational model. The first step is to formalize the notion of *store*, where constants and functions are defined.

Definition 1. *A* store *is a conditional* TRS $F = \langle \Sigma, \mathcal{R} \rangle$, *composed of a* signature Σ *and a set of* rules *(or* formulæ*)* $\mathcal{R}$. *A* signature Σ *is a set of function symbols (to simplify, we consider unsorted signatures in this paper). A* term *over* Σ *and variables* X *is either a variable* $x \in X$ *or an expression of the form* $f(t_1, \ldots, t_n)$, *where* $f \in \Sigma$ *is a function symbol of arity (number of arguments)* n *and the* t_i*'s are terms. We note* $T(\Sigma, X)$ *the set of* terms *over the signature* Σ *and variables* X. *We note* $\mathcal{V}ar(t)$ *the set of variables of* t. *A term* t *without variables, i.e.,* $\mathcal{V}ar(t) = \emptyset$ *is called* ground. *Rewrite Rules (elements of* $\mathcal{R}$*) are of the form*

$$l \rightarrow r \mid c$$

which has to be read: "l rewrites into r *if* c *holds", where* l *and* r *are terms, and* c *is a conjunction of equations of the form* $t_1 = t_2$ *such that* $\mathcal{V}ar(r) \cup \mathcal{V}ar(c) \subseteq \mathcal{V}ar(l)$

Furthermore, we use the function $eval(F, t)$, which evaluates the term t into its normal form with respect to the store $F = \langle \Sigma, \mathcal{R} \rangle$ (in this paper, we assume rewrite systems to be confluent and terminating). The actual (rewriting) strategy used by the operational semantics is not important in this paper. We say that equation $t_1 = t_2$ holds if the two terms have the same normal form.

We write $F \cup (l \rightarrow r \mid c)$ (resp. $F \setminus (l \rightarrow r \mid c)$) the store F to (resp. from) which the rule $l \rightarrow r \mid c$ has been added (resp. removed). We write $F \bullet (l \rightarrow r \mid c)$ the store equivalent to store F where all rules of left-hand side l (i.e. rules of the form $l \rightarrow r' \mid c'$) have been erased and rule $l \rightarrow r \mid c$ has been added.

Actions allow the modification of stores. We distinguish two kinds of actions: (i) *elementary actions* like assignment, addition or removal of rules, and (ii) *guarded actions* that are executed atomically only if their guards hold, providing high level synchronization.

Definition 2. *An* action α *is a pair consisting of a guard* g *and a sequence of elementary actions* a_i, *written:* $[g \Rightarrow \mathsf{a}_1; \ldots; \mathsf{a}_n]$. *A* guard *is a conjunction of equations whose validity in the store is decidable. The* elementary actions a *we consider in this paper are assignment* ($:=$), *addition of a rule to the store* (tell), *removal of a rule from the store* (del) *and* (skip) *the (elementary) action that does nothing.*

Basic processes in our model are succ (the process which terminates successfully), guarded actions α, or process calls $\mathrm{q}(t_1, \ldots, t_n)$. As usual in process algebra (see, e.g., [10]), we provide some operators for combining processes: parallel ($\|$) and sequential (;) composition as well as nondeterministic choice (+). Hence we have the following definition.

Definition 3. *Process definitions is provided by a set of guarded commands* $\Pi\!R = (\Pi, R_\Pi)$, *composed of a set of process signature* Π *and a set of guarded commands* R_Π. Π *is a set of process symbols. A* process term, p, *over* Π *is an expression defined by the following grammar:* $p ::= \mathsf{succ} \mid [g \Rightarrow \mathsf{a}] \mid p; p \mid p \parallel p \mid p + p \mid \mathrm{q}(t_1, \ldots, t_n)$ *where* $\mathrm{q} \in \Pi$ *and* $t_i \in T(\Sigma, X)$.
A process q *is defined by a sentence of the following form representing* m *nondeterministic guarded commands :*

$$\mathrm{q}(x_1, \ldots, x_n) \Leftarrow \textstyle\sum_{i=1}^{m} \alpha_i \,; p_i$$

where (for each i) α_i *is an action and* p_i *is a process term, such that the free variables of* α_i *and* p_i *are included in the parameter set* $\{x_1; \ldots; x_n\}$.

DEFINITION 4. *A* program *or a* component *K is a tuple $\langle F, \mathbb{R}\rangle$, where F is a store and $\mathbb{R}$ is a set of process definitions.*

The operational semantics of our execution model is defined by a transition system, defined by the rules shown in Fig. 1. The transition relation $\hookrightarrow$ describes the modification of the store by the execution of sequences of elementary actions. The execution (run) of processes is described by the transition relation $\longrightarrow$. Transitions are of the form $\langle F,p\rangle \longrightarrow \langle F',p'\rangle$ where F is a store and p is a process term. The relation $\longrightarrow$ is defined modulo the (classical) equivalence relation $\equiv_p$ which states that the operators $\|$ and $+$ are commutative and succ may vanish. Notice that the execution of actions is atomic (see rule (P_{guard})).

EXAMPLE 1. *As a first example, consider a direct translation of an example given in [2], showing how control flow can lead to information flow. It shows the limitations of the analysis of [30] in a concurrent context, and it is a simplified version of the example given in [29]. This example is studied, from a secrecy point of view in [7].*

$$\boxed{\begin{array}{lll}\alpha & \Leftarrow & [c_\alpha = \text{TT} \Rightarrow SPY := \text{TT};\ c_\beta := \text{TT}];\ \mathsf{succ}\\ \beta & \Leftarrow & [c_\beta = \text{TT} \Rightarrow SPY := \text{FF};\ c_\alpha := \text{TT}];\ \mathsf{succ}\\ \gamma & \Leftarrow & \big([PIN = \text{FF} \Rightarrow c_\alpha := \text{TT}]; \mathsf{succ}\big)+\\ & & \big([PIN = \text{TT} \Rightarrow c_\beta := \text{TT}]; \mathsf{succ}\big)\end{array}}$$

PIN, SPY, c_α, c_β, TT and FF are constants. c_α and c_β are initialized to FF. Notice that execution of $\alpha \parallel \beta \parallel \gamma$ copies the (secret) value PIN into the (public) constant SPY.

3. SECURITY POLICY

In this section we present how to declare a security policy by means of (confluent and terminating) rewrite systems. Normal forms of such security policies are elements of a finite lattice representing different privacy levels of data. Therefore for each function f we associate a set of rewrite rules of the form $f(\pi 1, x) \to \pi_2$, where π_1, and π_2 are elements of a lattice of privacy levels. At a first glance these rewrite rules may be seen as security profiles for functions. Such rewrite systems define a *security policy* modeling assumptions made by the programmer about the security behavior of functions. For instance it is sensible to declare that the result of the application of an encryption function onto private data is public. However, it is possible to define a security policy including the following rule $id(x) \to \bot$, where id stands for the classical identity function. In this case, the security policy is somewhat clumsy since $id(d)$, which is evaluated to the lowest privacy level, would be a public version of any information d (secret or not). The fact that a security policy is clumsy or not is out of the scope of this paper and thus not discussed here. The reader should notice that for each function f is associated (at least) two rewrite systems: a classical one dedicated to data processing (e.g., $f(x) \to x$) and a second one defining a security policy (e.g., $f(\top) \to \bot, f(\bot) \to \bot$).

Apart from security policies, our aim is to give a formal definition of secure processes (threads) in presence of a given security policy. For that we should distinguish authorized information leakage declared within a security policy, from unauthorized ones corresponding to classical interference issues. This is done through a specific evaluation process in which declassified terms are evaluated in a single store (therefore the evaluation of a declassified term will be the same for two different stores). We develop a notion of process equivalence accordingly. Then it is possible to state a non-interference like property (information may only flow from lower privacy levels to higher ones).

We start by recalling some technical definitions on rewrite systems.

3.1 Technical Preliminaries

We give in this subsection some technical definitions about rewriting that we use later. More details can be found for example in [4].

Let R be a rewrite system. A *substitution* is a mapping $\theta : X \to A$. Substitutions are extended to morphisms on terms by $\sigma(f(t_1,\ldots,t_n)) = f(\sigma(t_1),\ldots,\sigma(t_n))$ for every term $f(t_1,\ldots,t_n)$. An *occurrence* or *position* is a sequence of positive integers identifying a subterm in a term. For every term t, the empty sequence denoted by Λ, identifies t itself. For every term of the form $f(t_1,\ldots,t_k)$, the sequence $i \cdot p$, where i is a positive integer not greater than k and p is a position, identifies the subterm of t_i at p. Positions are ordered by prefix order, that is a position p is greater than position p' if p' is a prefix of p. The subterm of t at p is denoted by $t|_p$ and the result of *replacing* $t|_p$ with s in t is denoted by $t[s]_p$. A *reduction step* is an application of a rewrite rule to a term, i.e., $t \to_{p,\rho} s$ if there exist a position p, a rewrite rule $\rho = l \to r \mid c$ and a substitution σ with $t|_p = \sigma(l)$, $\sigma(c)$ holds and $s = t[\sigma(r)]_p$. In this case we say t is *rewritten* (at position p) *to* s. We will omit the subscripts p and ρ if they are clear from the context. We may also write $t \to_R s$ or $t \to s$. We write $\xrightarrow{*}_R$ the reflexive and transitive closure of $\to_R$. $\xleftrightarrow{*}_R$ is the reflexive, symmetric and transitive closure of $\to_R$.

A term t is *reducible to* a term s if $t \xrightarrow{*} s$. A term t is called *irreducible* or in *normal form* if there is no term s with $t \to s$. A term s is a *normal form of* t if t is reducible to the irreducible term s.

A TRS R is confluent if for all terms t_1, t_2 and t_3 such that $t_1 \xrightarrow{*}_R t_2$ and $t_1 \xrightarrow{*}_R t_3$, there exists a term t_4 such that $t_2 \xrightarrow{*}_R t_4$ and $t_3 \xrightarrow{*}_R t_4$. R is terminating if there exists no endless derivation $t \to t_1 \to \ldots$. When the considered rewrite system is confluent and terminating, every term t has a unique normal form which we write $t!_R$.

In the following definition we recall the notion of descendant. This notion will be used in order to take into account declared information leakage.

DEFINITION 5. *Let $A = t \to_{u,l\to r} t'$ be a reduction step of some term t into t' at position u with rule $l \to r$. The set of* descendants *(or* residuals*) of a position v by A, denoted $v \setminus A$, is*

$$v \setminus A = \begin{cases} \emptyset & \textit{if } v = u\cdot p \textit{ and}\\ & l|_p \textit{ is not a variable,}\\ \{v\} & \textit{if } u \not\leq v,\\ \{u \cdot p' \cdot q \mid r|_{p'} = x\} & \textit{if } v = u \cdot p \cdot q \textit{ and}\\ & l|_p = x, \textit{ and } x \textit{ variable.}\end{cases}$$

The set of descendants *of a position v by a reduction sequence B is defined by induction as follows*

$$v \setminus B = \begin{cases} \{v\} & \textit{if } B \textit{ is the null derivation,}\\ \displaystyle\bigcup_{w \in v\setminus B'} w \setminus B'' & \textit{if } B = B'B'', \textit{ where } B' \textit{ is}\\ & \textit{the initial step of } B.\end{cases}$$

$$\langle F, \mathsf{tell}(R); a\rangle \hookrightarrow \langle F \cup R, a\rangle \quad (ea_{\mathsf{tell}}) \qquad \langle F, f := t; a\rangle \hookrightarrow \langle F \bullet (f \rightarrow eval(F,t)), a\rangle \quad (ea_{:=})$$

$$\langle F, \mathsf{del}(R); a\rangle \hookrightarrow \langle F \setminus R, a\rangle \quad (ea_{\mathsf{del}}) \qquad \langle F, \mathsf{skip}; a\rangle \hookrightarrow \langle F, a\rangle \quad (ea_{\mathsf{skip};})$$

$$\mathsf{succ}; p \equiv_p p \quad (Eq_{\mathsf{succ};}) \qquad p_1 + p_2 \equiv_p p_2 + p_1 \quad (Eq_{+\ com})$$

$$\mathsf{succ} \parallel p \equiv_p p \quad (Eq_{\mathsf{succ}\parallel}) \qquad p_1 \parallel p_2 \equiv_p p_2 \parallel p_1 \quad (Eq_{\parallel\ com})$$

$$\frac{p_1 \equiv_p p_2 \quad \langle F, p_2\rangle \longrightarrow \langle F', p_3\rangle \quad p_3 \equiv_p p_4}{\langle F, p_1\rangle \longrightarrow \langle F', p_4\rangle} \qquad (P_{\equiv_p})$$

$$\frac{\langle F, a_1; \ldots; a_n; \mathsf{skip}\rangle \hookrightarrow^* \langle F', \mathsf{skip}\rangle \quad eval(F, t_i) = eval(F, t'_i)}{\langle F, [g \Rightarrow a_1; \ldots; a_n]\rangle \longrightarrow \langle F', \mathsf{succ}\rangle} \quad g = \bigwedge_{i=1}^{m} t_i = t'_i \qquad (P_{guard})$$

$$\frac{(\mathrm{q}(x_1, \ldots, x_n) \Leftarrow \Sigma_{j=1}^{m} \alpha_j\, ; p_j) \in \Pi\!R \quad \langle F, (\Sigma_{j=1}^{m} \alpha_j\, ; p_j)[t_i/x_i]\rangle \longrightarrow \langle F', p'\rangle}{\langle F, \mathrm{q}(t_1, \ldots, t_n)\rangle \longrightarrow \langle F', p'\rangle} \qquad (P_{abs})$$

$$\frac{\langle F, p_1\rangle \longrightarrow \langle F', p'_1\rangle}{\langle F, p_1 + p_2\rangle \longrightarrow \langle F', p'_1\rangle} \quad (P_+) \qquad \frac{\langle F, p_1\rangle \longrightarrow \langle F', p'_1\rangle}{\langle F, p_1 \ \mathsf{op}\ p_2\rangle \longrightarrow \langle F', p'_1 \ \mathsf{op}\ p_2\rangle} \quad \mathsf{op} \in \{\parallel, ;\} \qquad (P_{\mathsf{op}})$$

Figure 1: Inference Rules Defining the Operational Semantics

A position uniquely identifies a subterm of a term. The notion of *descendant* for terms stems directly from the corresponding notion for positions.

3.2 Security Policy

We recall the basic notion of privacy levels. Formally, privacy levels are elements of a lattice $\mathfrak{L}$. We note $\sqsubseteq$ the order defined on $\mathfrak{L}$ and make no difference between the lattice and the set of its elements, i.e., $\mathfrak{L}$ denotes in the same time the lattice and its carrier. We write respectively $\bot$ and $\top$ the bottom and the top element of $\mathfrak{L}$. If π_1, and π_2 are two elements of $\mathfrak{L}$ and $\pi_1 \sqsubseteq \pi_2$, we say that π_2 is *more private* than π_1. We write $\sqcup$ for the join operation (least upper bound) and $\sqcap$ for the meet operation (greatest lower bound).

In the following we suppose given such a finite lattice of privacy levels $\mathfrak{L}$.

Definition 6 (Security policy). *A security policy, $\mathcal{SP}$, is a terminating and confluent TRS defined over the signature $\Sigma \cup \mathfrak{L}$ such that*

- *$\mathcal{SP}$ introduces no junk into $\mathfrak{L}$. I.e., for all ground terms, t, over $\Sigma \cup \mathfrak{L}$, $t!_{\mathcal{SP}}$ is in $\mathfrak{L}$.*
- *$\mathcal{SP}$ introduces no confusion into $\mathfrak{L}$. I.e., $\forall \tau_1, \tau_2 \in \mathfrak{L}, \tau_1 \neq \tau_2 \implies \tau_1 \not\xrightarrow{*}_{SP} \tau 2$*
- *functions in Σ are monotonic w.r.t. privacy levels (i.e., $\forall \tau_1, \ldots, \tau_n \in \mathfrak{L}, \forall \tau'_i \in \mathfrak{L}, \tau_i \sqsubseteq \tau'_i \implies$
$f(\tau_1, \ldots, \tau_i, \ldots, \tau_n)!_{\mathcal{SP}} \sqsubseteq f(\tau_1, \ldots, \tau'_i, \ldots, \tau_n)!_{\mathcal{SP}}$)*

The no junk condition ensures that every data represented as a (ground) term has a privacy level ($\in \mathfrak{L}$). The no confusion property guaranties that different privacy levels will never be equated by mistake. The third property is not mandatory but rather a sensible one. It also simplifies some definitions such as the notion of privacy level of a rule given later on in the paper.

Consider the following example. Let $\Sigma = \{f, g, h\}$ be a signature consisting of three function symbols with the following respective arities 2, 2 and 1. Let $\mathfrak{L} = \{\bot, \top\}$ and $\bot \sqsubseteq \top$. The following set of rules defines a security policy $\mathcal{SP}$:

$$f(x,y) \rightarrow \top \qquad g(x) \rightarrow x$$
$$h(\top, x) \rightarrow \bot \qquad h(\bot, x) \rightarrow \bot$$

Now $g(f(\top, \bot))$ has $\top$ as normal form whereas $g(h(\top, \top))$ has $\bot$ as normal form.

Usually the privacy level is supposed to increase through computation in order to avoid information leakage. Therefore the evaluation of a function using information of privacy level π should yield a result with a privacy level *greater* than or equal to π. However, this is no longer the case whenever one has to deal with programs which downgrade or declassify data. In the previous example of security policy $\mathcal{SP}$, function h declassifies information. Indeed the rule $h(\top, x) \rightarrow \bot$ depends on $\top$ privacy level and gives a result of level $\bot$. It means that this security policy *allows* function h to declassify the information given in its first argument.

Declassifying functions model the ways offered to a programmer to make interference between high level and low level data. In some sense they can be seen as cryptographic functions: the result of a encryption of data can be made public.

One feature of our approach is that it can define subtle security policies. For instance, there are several levels in the power of cyphers, some are unbreakable (Vernam cipher) whereas others offer less security (Caesar cipher). There is also the case of algorithms depending on the size of the used key: an RSA with a 100-digits key is more reliable than an RSA with 50-digits key. It is possible to encode such subtleties within adequate lattice and security policy. Think for instance of a function RSA of arity 2, where the first argument is the key and the second the message to be cyphered. Now consider the following lattice of privacy levels $\mathfrak{L} = \{\bot, \top, \top_1, K_{50}, K_{100}\}$.

$$\begin{tikzcd} & \top \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ K_{50} \arrow[dr, dash] & \top_1 \arrow[d, dash] & K_{100} \arrow[dl, dash] \\ & \bot & \end{tikzcd}$$

The idea behind this definition is that K_{50}, K_{100} represents privacy levels dedicated to keys for encryption algorithms whereas $\top, \top_1$ and $\bot$ are privacy level for data to be encrypted. We could then give the following security policy:

$$\begin{gathered} RSA(K_{100}, y) \rightarrow \bot \\ RSA(K_{50}, x) \rightarrow \top_1 \\ RSA(\top_1, x) \rightarrow x \\ RSA(\top, x) \rightarrow x \quad RSA(\bot, x) \rightarrow x \end{gathered}$$

Notice that the three last rules are given in order to comply with definition 6. These rules are not supposed to be used in practice since RSA's first argument, the encryption key, has to be of privacy level either K_{50} or K_{100}.

In other words, encryption with key K_{100} downgrades everything to public level ($\bot$), whereas encryption with a less powerful key downgrades to a more private level ($\top_1$), because the cypher is less powerful. The other rules are given for the sake of completeness (security policy has to be terminating for any argument and the result must be an element of $\mathfrak{L}$ see definition 6). In an actual setting they are of no use since the first argument of an encryption algorithm is meant to be a key, and thus must have a privacy level in $\{K_{50}, K_{100}\}$. This must be checked in an actual system but it is not meaningful here.

In the rest of this section we suppose given, although non specified, a security policy $\mathcal{SP}$.

Definition 7 (Privacy level of a term). *Let t be a term. The privacy level of t w.r.t. a security policy $\mathcal{SP}$, written $\pi(t)$ is the normal form $\sigma(t)!_{SP}$ where the substitution σ instantiates every variable of t by $\bot$.*

In our previous example the privacy level of $f(g(h(x, y)), z)$ is thus the normal form of $f(g(h(\bot, \bot)), \bot)$ which is $\top$. Notice that since $\mathcal{SP}$ defines monotonic functions (see definition 6), the privacy level of a term is the minimal privacy level obtained for any ground instance of this term.

We extend the notion of privacy level to rewrite rules by saying that the privacy level of a rule $l \rightarrow r \mid c$ is equal to the privacy level of its left-hand side $\pi(l)$. We extend the notion of privacy to conjunction of equations of the form $t_1 = t_2$ by:

$$\pi(t_1 = t_2 \wedge ... \wedge t_n = t_{n-1}) = \bigsqcap_{i=1}^{n} \pi(t_i)$$

We also extend these definitions to actions as follows:

$$\begin{gathered} \pi(\mathsf{skip}) = \top \qquad \pi(f := t) = \pi(f) \\ \pi(\mathsf{tell}(l \rightarrow r \mid c)) = \pi(\mathsf{del}(l \rightarrow r \mid c)) = \pi(l) \\ \pi([g \Rightarrow a_1; \ldots; a_n]) = \bigsqcap_{i=1}^{n} \pi(a_i) \end{gathered}$$

The intuitive explanation of those definitions is that the privacy level of an action is the greatest lower bound of its components. That is, if an action as a privacy level π then, at the worst, it can disclose information of this privacy level (see definition 9 for further justifications of tell and del actions).

We now introduce the notion of declassified terms. As we said earlier a function having a security policy assigning a level less private than its argument *may* declassify information. Indeed, in some ways it depends on private data and gives a result of a more public status. Therefore, they represent potential information flows. We identify terms potentially declassified, i.e. terms which are made public through the use of declassifying functions.

Definition 8 (Declassified term). *Let $t = f(t_1, \ldots, t_n)$ with $n \geq 1$ be a term. It is said to be* declassified *if:*

1. *There is $j \in \{1, \ldots, n\}$ such that $\pi(t_j) \not\sqsubseteq \pi(t)$, or*
2. *t is a subterm of a declassified term.*

Consider the following term $t = h(f(k), g(f(k)))$ together with the following security policy

$$\begin{array}{ll} k \rightarrow \top & f(x) \rightarrow \top \\ g(x) \rightarrow \bot & h(x, y) \rightarrow \top \end{array}$$

Terms $t_{|1}$ is not declassified, $t_{|2}$, $t_{|2.1}$ and $t_{|2.1.1}$ are declassified. Notice that $t_{|1}$ and $t_{|2.1}$ represent the same data but have not the same status with respect to declassification.

3.3 Secure Processes

We now address the problem of secure process terms with respect to a given security policy. We informally recall our aim: we want to abstract from information flows declared in the security policy, typically flows coming from declassified terms, but want to prohibit other ones.

We start by the definition of a notion of well defined rewrite rules. The idea is that there should be no information leakage from left-hand side to right-hand side, and thus the privacy level may only decrease through rewrite rule applications. That is to say, the computation of the normal form of a term with privacy level π uses only terms the privacy levels of which are less than or equal to π.

Definition 9 (Well defined rewrite rules). *A rewrite rule $l \rightarrow r \mid c$ is* well defined *whenever the following conditions hold:*

1. *$\pi(\sigma(r)) \sqsubseteq \pi(\sigma(l))$ and $\pi(\sigma(c)) \sqsubseteq \pi(\sigma(l))$ for all ground substitutions $\sigma : \mathcal{V}ar(l) \rightarrow \mathfrak{L}$*
2. *l is a declassified term whenever r contains declassified subterms.*

Indeed, rules of the form $SPY \rightarrow PIN$, given a security policy such that $SPY \rightarrow \bot, PIN \rightarrow \top$ is not acceptable since it directly discloses the value of a private information. The same remark can be done, altough more subtly, for rules of the form $SPY \rightarrow true \mid PIN = true$. Condition (2) is not mandatory. It ensures that within a rewrite derivation a descendant of a (sub-)term t is declassified only if t is itself declassified. This property allows us to give a simplified definition of what we call declassified evaluation (cf. Definition 11).

In the following we only consider stores having well defined rewrite rules.

We now introduce a notion of store equivalence up to some privacy level π. Informally two stores are equivalent up to π if they agree on rules with privacy level no greater than π.

Definition 10 (Store equivalence). *Let F_0, F_1 be two stores, and $\pi \in \mathfrak{L}$. We say that F_0 and F_1 are π-equivalent $F_0 \cong_\pi F_1$ iff for all $i \in \{0,1\}$, for all rules $\rho_i = l_i \to r_i \mid c_i \in \mathcal{R}_i$ such that $\pi(\rho_i) \sqsubseteq \pi$, there exists a rule $\rho_{1-i} = l_{1-i} \to r_{1-i} \mid c_{1-i} \in \mathcal{R}_{1-i}$, with $\rho_i = \rho_{1-i}$, up to variable renaming.*

The following definition is the key to handle formally declared information leakage via a security policy (see Definitions 13 and 14). In this definition, we propose to evaluate declassified terms w.r.t. a specific store.

Definition 11 (Declassified Evaluation). *Let F, F' be two stores. $\mathfrak{s}\_eval(F, F', t)$ is the normal form of t computed using the rewrite relation $\overset{F,F'}{\to}$ defined as follows: $t \overset{F,F'}{\to}_{u,l\to r|c} t'$ such that*

- *$l \to r \mid c \in F'$ If the position u is a descendant of a declassified term,*
- *$l \to r \mid c \in F$, otherwise.*

In other terms $\mathfrak{s}\_eval(F, F')$ is the evaluation process which computes declassified parts (and their descendants) of a term using rules in F' and the other parts using rules in F. If we make the assumption that both stores are confluent, the result is also confluent because of the condition on well-defined rules stating that right hand sides may contain declassified terms only if the left hand side is a declassified term (see definition 9). This way all declassified subterms are descendant of a declassified subterm already present in the term to normalize.

We now define a *declassified operational semantics* based on this declassified evaluation.

Definition 12 (Declassified Op. Sem.). *Let F, F' be two stores, and p a process term. We define the declassified operational semantics as a transition relation $\xrightarrow{F'}$. Transitions are of the form $\langle F, p\rangle \xrightarrow{F'} \langle F'', p'\rangle$. Rules are defined as the ones of Fig. 1, except for rules $(ea_{:=})$ and (P_{guard}) where $eval(F,t)$ is replaced by $\mathfrak{s}\_eval(F, F', t)$.*

Now, we are almost ready to define a notion of secure process term w.r.t. a security policy. For that, we introduce a notion of secure process term up to some privacy level π. A process term will be said secure up to π if its *behaviour*, limited to the effect done on data having a privacy level less than or equal to π, does not depend on data with a privacy level higher than π.

Definition 13 (Secure process term up to π). *Let p be a ground process term, $\pi \in \mathfrak{L}$. p is secure up to privacy π, and two π-equivalent stores F_1, F_2 if $\langle F_1, p\rangle \longrightarrow \langle F_1', p'\rangle$ implies that:*

- *either $\exists F_2'$ such that $\langle F_2, p\rangle \xrightarrow{F_1} \langle F_2', p'\rangle$, $F_1' \cong_\pi F_2'$ and p' is secure up to privacy π and stores F_1', F_2'.*
- *or $\langle F_2, p_2\rangle \overset{F_1}{\not\longrightarrow}$ and for all $F_1^\sharp, p_1^\sharp$ such that $\langle F_1', p_1'\rangle \longrightarrow^* \langle F_1^\sharp, p_1^\sharp\rangle$ we have $F_1^\sharp \cong_\pi F_2$.*

Thus, a process term is secure up to level π if within its runs data with privacy level higher than π do not influence data of a privacy status lower than π (stores stay π-equivalent) except when this leakage is authorized by the security policy. We have modeled the hypothesis done by the security policy by evaluating declassified terms on a single store. Note that the apparent distinction between F_1 and F_2 (the definition is not symmetric) is not significant since Definition 14 must hold for any appropriate equivalent stores F_1, F_2.

We are now able to define what means to be safe with respect to a given security policy. A process term is safe whenever it is secure for all privacy levels of $\mathfrak{L}$.

Definition 14 (Secure ground process term). *A ground process term p of a component $\langle F, \mathrm{I\!R}\rangle$ is* secure *w.r.t. a security policy $\mathcal{SP}$, iff for all π, all stores F_1, F_2 such that $F \cong_\pi F_1 \cong_\pi F_2$, p is secure up to privacy level π and stores F_1, F_2.*

We now give an example involving encrypted communications to illustrate these definitions.

Example 2 (Encrypted communication). *Suppose that we have two functions (with arity one for sake of simplicity) crypt and decrypt for encryption/decryption. Let us consider the following security policy: $crypt(x) \to \bot$ and $decrypt(x) \to \top$ which means that encrypted data are considered as public ($\bot$) and decrypted data are secret ($\top$). Then the communication of a secret through a public channel between two processes may be written in the following way:*

$$\boxed{\begin{array}{lcl} \alpha & \Leftarrow & \left[c_\alpha = \mathrm{TT} \Rightarrow \begin{array}{l} pub := crypt(PIN)\,; \\ done := \mathrm{TT}\,;\ c_\alpha := \mathrm{FF} \end{array}\right];\ \alpha' \\ \\ \beta & \Leftarrow & \left[done = \mathrm{TT} \Rightarrow \begin{array}{l} k := decrypt(pub)\,; \\ done := \mathrm{FF} \end{array}\right];\ \beta' \end{array}}$$

with the security policy : $pub \to \bot, c_\alpha \to \bot, done \to \bot$ and $PIN \to \top, k \to \top$. Then, starting from a store where done and c_α are two constants defined by the following rewrite rules $done \to \mathrm{FF}, c_\alpha \to \mathrm{TT}$, the process term $\alpha \parallel \beta$ communicates the secret PIN through a public channel pub by means of an encryption function.

The point of this example is the following. Imagine that a third process, say a spy, γ runs in parallel with α, β. Then γ has access to pub and may use it to modify its low level behavior. Therefore high level data may interfere with low level behavior. For instance if γ is defined as

$$\boxed{\begin{array}{lcl} \gamma & \Leftarrow & [pub = 1 \Rightarrow y := 1] \\ & & [pub \neq 1 \Rightarrow y := 2] \end{array}}$$

with y such that $y \to \bot$ in the security policy, then $\alpha \parallel \beta \parallel \gamma$ may formally exhibit information flows from high to low level. Indeed for two $\bot$-equivalent stores the value of y may differ, thus there are observable differencies from a $\bot$-level point of view. It suffices for this to take two stores where the value of $encrypt(pin)$ is different.

Nevertheless, this should be acceptable in such a case since data exchanged on pub is encrypted, and since the security *policy assigns $\perp$ privacy level to encrypted terms. The reader may check that $\alpha \parallel \beta \parallel \gamma$ is a secure ground process term.*

On the other hand, notice that, due to security policy, once decrypted (here the result is stored in the high level constant k) the status of data gets back to high level, hence cannot be used inappropriately later.

4. SECURE PROCESS ANALYSIS

In section 4.1, we present an abstract interpretation on which our analysis algorithm is based. The latter consists in producing a set of inequations over privacy levels. This algorithm is presented in section 4.2. In section 4.3 we show how the satisfaction of inequations is related to our definition of secure process terms and focus on points induced by security policies.

4.1 Abstract interpretation

We define an abstract operational semantics for a component K, and security policy $\mathcal{SP}$. The abstraction of a store is a set of inequations over privacy levels, more precisely over terms of $\mathcal{SP}$. The abstract operational semantics defines new runs of abstract processes. These runs collect inequations over $\mathcal{SP}$ terms. Since the number of these inequations is finite, it is possible to produce the whole set of inequations for a process term. We prove in Theorem 1 that if a security policy $\mathcal{SP}$ defined on K is such that all inequations hold, then the analyzed process term is secure in the sense of definition 14. That is to say, no unwanted interference happens. This is a kind of abstract interpretation, but do not introduce all the machinery of [3].

We define *privacy formulæ* $\mathfrak{f}$ by: $\mathfrak{f} ::= \pi \mid c \mid t \mid \mathfrak{f} \sqcap \mathfrak{f} \mid \mathfrak{f} \sqcup \mathfrak{f}$ where c denotes a constant and t a term of $\mathcal{SP}$. *Privacy inequations* are statements of the form $\mathfrak{f}_1 \sqsubseteq \mathfrak{f}_2$.

DEFINITION 15. *Let $F = \langle \Sigma, \mathcal{R} \rangle$ be a store, we define $F^{\mathcal{A}} = \langle \Sigma^{\mathcal{A}}, \mathcal{R}^{\mathcal{A}} \rangle$, its* abstract store*, as a set of privacy inequations defined as follows: for all rules $l \rightarrow r \mid c$ in $\mathcal{R}$, there are inequations $r \sqsubseteq l, c \sqsubseteq l$ in $\mathcal{R}^{\mathcal{A}}$ and $\mathcal{R}^{\mathcal{A}}$ contains only those rules. $\Sigma^{\mathcal{A}}$ is the signature defining the same symbols with the same arity than in Σ but is single sorted.*

We now define a notion of compatibility between an abstract store $F^{\mathcal{A}}$ and a security policy.

DEFINITION 16. *An abstract store $F^{\mathcal{A}}$ is compatible with a security policy $\mathcal{SP}$ iff all inequations in $F^{\mathcal{A}}$, say $t \sqsubseteq t'$, are valid in $\mathfrak{L}$. That is to say, for all ground substitution $\mathfrak{s} : \mathcal{V}ar(t) \cup \mathcal{V}ar(t') \rightarrow \mathfrak{L}$, $\mathfrak{s}(t)!_{\mathcal{SP}} \sqsubseteq \mathfrak{s}(t')!_{\mathcal{SP}}$.*

We use an abstract execution to collect constraints that ensure security of process terms. Informally, the abstract operational semantics is defined by a transition system, the states of which are triples $\langle F^{\mathcal{A}}, p^{\mathcal{A}}, \tau \rangle$ consisting of an abstract store $F^{\mathcal{A}}$, an abstract process term $p^{\mathcal{A}}$ and a privacy formula τ, corresponding to the highest level checked in a guard up to the current point in the execution. Abstract transitions generate constraints, depending on τ as well as on the privacy level of terms manipulated, and record them into the abstract store. Fig. 2 gives the rules defining the abstract transition relation

Abstract execution of sequences of elementary actions is described by the relation $\hookrightarrow^{\mathcal{A}}$. Abstract elementary actions modify an abstract store $F^{\mathcal{A}}$ with respect to a privacy level τ of $\mathfrak{L}$. Since in a parallel composition $p \parallel q$, p might check guards of high level while q only works on low privacy levels, we need to duplicate the privacy level τ, in order to not reject such processes (as constraints generated by p can be too strong compared to q). Hence, we introduce *abstract operators* $+^{\mathcal{A}}$, $\|^{\mathcal{A}}$, $;^{\mathcal{A}}$and *abstract process terms* (or $\mathcal{M}$-terms), defined by the grammar: $\mathcal{M} ::= \langle p, \tau \rangle \mid \mathcal{M} \|^{\mathcal{A}} \mathcal{M} \mid \mathcal{M} +^{\mathcal{A}} \mathcal{M} \mid \mathcal{M} ;^{\mathcal{A}} \mathcal{M}$. In order to translate concrete to abstract operators, we define a *transformation relation* $\longmapsto$. It is clear that $\longmapsto$ is confluent and strongly normalizing. We write $\overline{\mathcal{M}}^{\mathsf{nf}}$ the normal form of $\mathcal{M}$ w.r.t. $\longmapsto$.

By inspection of the similarities of the inference rules in Figs. 1 and 2 we can prove that to each concrete transition step corresponds an abstract reduction step.

LEMMA 1. *If $\langle F, \mathsf{a} ; a \rangle \hookrightarrow \langle F', a \rangle$ then, for all τ and constraint sets C, there exists $C' \supseteq C$ such that*

$$\langle F^{\mathcal{A}} \cup C, \langle \mathsf{a} ; a, \tau \rangle \rangle \hookrightarrow^{\mathcal{A}} \langle F'^{\mathcal{A}} \cup C', \langle a, \tau \rangle \rangle$$

Let ϕ be a function which associates to an abstract process term $\mathcal{M}$ a corresponding concrete process term by omitting all privacy levels (i.e., $\phi(\langle p, \tau \rangle) = p$, $\phi(\mathcal{M} \|^{\mathcal{A}} \mathcal{M}') = \phi(\mathcal{M}) \parallel \phi(\mathcal{M}')$, $\phi(\mathcal{M} +^{\mathcal{A}} \mathcal{M}') = \phi(\mathcal{M}) + \phi(\mathcal{M}')$, $\phi(\mathcal{M} ;^{\mathcal{A}} \mathcal{M}') = \phi(\mathcal{M}) ; \phi(\mathcal{M}')$).

LEMMA 2. *Let $K = \langle F, \Pi\mathcal{R} \rangle$ and p a process term of K. If $\langle F, p \rangle \longrightarrow \langle F', p' \rangle$ then for all $\mathcal{M}$ such that $\phi(\mathcal{M}) = p$ and for all constraint sets C there exist $\mathcal{M}'$, C' such that $\langle F^{\mathcal{A}} \cup C, \overline{\mathcal{M}}^{\mathsf{nf}} \rangle \longrightarrow^{\mathcal{A}} \langle F'^{\mathcal{A}} \cup C', \mathcal{M}' \rangle$, $\phi(\mathcal{M}') \equiv_p p'$ and $C \subseteq C'$.*

4.2 Process Term Analysis

The idea of our analysis is to collect the constraints computed by *all* possible abstract executions of a ground process term, say p. We claim that if the collected constraints are all valid in $\mathfrak{L}$, then the process term, p, is secure for the considered security policy. Crucial for the termination of our analysis is that the abstract store becomes stable during an abstract execution, i.e., after a certain point no more new privacy inequations are created.

DEFINITION 17. *The* analysis reduction $\rightsquigarrow$ *is the relation between triples of the form $\langle F^{\mathcal{A}}, \mathcal{M}, \mathfrak{H} \rangle$, where $\mathfrak{H}$ (denoting the $\mathfrak{H}$istory of executed process calls) is a set of pairs of the form $\langle \mathrm{q}, [\pi(t_1) ; \ldots ; \pi(t_n)] \rangle$. $\rightsquigarrow$ is defined as follows:*

- *if $\langle F^{\mathcal{A}}, \mathcal{M} \rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}' \rangle$ using a reduction rule different from $(\mathsf{A}P_{abs})$, then $\langle F^{\mathcal{A}}, \mathcal{M}, \mathfrak{H} \rangle \rightsquigarrow \langle F^{\mathcal{A}\prime}, \mathcal{M}', \mathfrak{H} \rangle$ and*
- *if $\mathcal{M} = \langle \mathrm{q}(t_1, \ldots, t_n), \tau \rangle$ and $\langle F^{\mathcal{A}}, \langle (\tau_{j=1}^{m} \alpha_j ; p_j)[x_i/t_i], \tau \rangle \rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}' \rangle$, where the process q is defined by $\left(\mathrm{q}(x_1, \ldots, x_n) \Leftarrow \tau_{j=1}^{m} \alpha_j ; p_j \right) \in \Pi\mathcal{R}$, then*

$$\langle F^{\mathcal{A}}, \mathcal{M}, \mathfrak{H} \rangle \rightsquigarrow \begin{cases} \langle F^{\mathcal{A}}, \langle \mathsf{succ}, \tau \rangle, \mathfrak{H} \rangle & \text{if } \langle \mathrm{q}, [\pi(t_1); \\ & \quad \ldots ; \pi(t_n)] \rangle \in \mathfrak{H} \\ & \\ \langle F^{\mathcal{A}\prime}, \mathcal{M}', \mathfrak{H} \rangle \cup & \text{otherwise} \\ \langle \mathrm{q}, [\pi(t_1); \ldots ; \pi(t_n)] \rangle & \end{cases}$$

Using the fact that the number of non equivalent abstract process calls is finite (since $\sqcup$ is idempotent and associative), we can prove that there are no infinite $\rightsquigarrow$ reduction

$$\langle F^{\mathcal{A}}, f := t; a, \tau\rangle \hookrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}} \cup \{t \sqsubseteq f\} \cup \{\tau \sqsubseteq f\}, a, \tau\rangle \qquad (\mathsf{A}ea_{:=})$$

$$\langle F^{\mathcal{A}}, \mathsf{tell}(l \to r \mid c); a, \tau\rangle \hookrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}} \cup \{r \sqsubseteq l\} \cup \{c \sqsubseteq l\} \cup \{\tau \sqsubseteq l\}, a, \tau\rangle \qquad (\mathsf{A}ea_{\mathsf{tell}})$$

$$\langle F^{\mathcal{A}}, \mathsf{del}(l \to r \mid c); a, \tau\rangle \hookrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}} \cup \{\tau \sqsubseteq l\}, a, \tau\rangle \qquad (\mathsf{A}ea_{\mathsf{del}})$$

$$\langle F^{\mathcal{A}}, \mathsf{skip}; a, \tau\rangle \hookrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}}, a, \tau\rangle \qquad (\mathsf{A}ea_{\mathsf{skip};})$$

$$\frac{p_1 \equiv_p p_2}{\langle p_1, \tau\rangle \equiv^{\mathcal{A}} \langle p_2, \tau\rangle} \qquad (\mathsf{A}Eq_{\equiv_p})$$

$$\mathcal{M}_1 \|^{\mathcal{A}} \mathcal{M}_2 \equiv^{\mathcal{A}} \mathcal{M}_2 \|^{\mathcal{A}} \mathcal{M}_1 \qquad (\mathsf{A}Eq_{\|^{\mathcal{A}}})$$

$$\mathcal{M}_1 +^{\mathcal{A}} \mathcal{M}_2 \equiv^{\mathcal{A}} \mathcal{M}_2 +^{\mathcal{A}} \mathcal{M}_1 \qquad (\mathsf{A}Eq_{+^{\mathcal{A}}})$$

$$\langle p_1 \ \mathsf{op}\ p_2, \tau\rangle \longmapsto \langle p_1, \tau\rangle \ \mathsf{op}^{\mathcal{A}} \ \langle p_2, \tau\rangle \quad \mathsf{op} \in \{\|, ;, +\} \qquad (\mathsf{Aop}^{\mathcal{A}}\text{-}I)$$

$$\langle \mathsf{succ}, \tau_1\rangle \|^{\mathcal{A}} \langle \mathsf{succ}, \tau_2\rangle \longmapsto \langle \mathsf{succ}, \tau_1 \sqcup \tau_2\rangle \qquad (\mathsf{A}\|^{\mathcal{A}}\text{-}E)$$

$$\langle \mathsf{succ}, \tau_1\rangle \ ;^{\mathcal{A}} \langle p_2, \tau_2\rangle \longmapsto \langle p_2, \tau_1 \sqcup \tau_2\rangle \qquad (\mathsf{A}\,;^{\mathcal{A}}\text{-}E)$$

$$\frac{\overline{\mathcal{M}_1}^{\mathsf{nf}} \equiv^{\mathcal{A}} \mathcal{M}_2 \quad \langle F^{\mathcal{A}}, \mathcal{M}_2\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}_3\rangle \quad \overline{\mathcal{M}_3}^{\mathsf{nf}} \equiv^{\mathcal{A}} \mathcal{M}_4}{\langle F^{\mathcal{A}}, \mathcal{M}_1\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}_4\rangle} \qquad (\mathsf{A}P_{\equiv^{\mathcal{A}}})$$

$$\frac{\langle F^{\mathcal{A}}, \langle a_1; \ldots; a_n; \mathsf{skip}, \tau \sqcup g\rangle\rangle \hookrightarrow^{\mathcal{A}} {*} \langle F^{\mathcal{A}\prime}, \langle \mathsf{skip}, \tau \sqcup g\rangle\rangle}{\langle F^{\mathcal{A}}, \langle [g \Rightarrow a_1; \ldots; a_n], \tau\rangle\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \langle \mathsf{succ}, \tau \sqcup g\rangle\rangle} \qquad (\mathsf{A}P_{guard})$$

$$\frac{(\mathrm{q}(x_1, \ldots, x_n) \Leftarrow \tau_{j=1}^{m} \alpha_j \,; p_j) \in \Pi\!\!R \quad \langle F^{\mathcal{A}}, \langle (\tau_{j=1}^{m} \alpha_j \,; p_j)[t_i/x_i], \tau\rangle\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}'\rangle}{\langle F^{\mathcal{A}}, \langle \mathrm{q}(t_1, \ldots, t_n), \tau\rangle\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}'\rangle} \qquad (\mathsf{A}P_{abs})$$

$$\frac{\langle F^{\mathcal{A}}, \mathcal{M}_1\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}_1'\rangle}{\langle F^{\mathcal{A}}, \mathcal{M}_1 \mathsf{op}^{\mathcal{A}} \mathcal{M}_2\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}_1' \mathsf{op}^{\mathcal{A}} \mathcal{M}_2\rangle} \quad \mathsf{op} \in \{;, \|\} \qquad (\mathsf{A}P_{\mathsf{op}^{\mathcal{A}}})$$

$$\frac{\langle F^{\mathcal{A}}, \mathcal{M}_1\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}_1'\rangle}{\langle F^{\mathcal{A}}, \mathcal{M}_1 +^{\mathcal{A}} \mathcal{M}_2\rangle \longrightarrow^{\mathcal{A}} \langle F^{\mathcal{A}\prime}, \mathcal{M}_1'\rangle} \qquad (\mathsf{A}P_{+^{\mathcal{A}}})$$

Figure 2: Abstract Operational Semantics

sequences. Since, in addition, the number of rules one may apply is always finite, we can define the result returned by our analysis as the collection of all reachable abstract stores. See [5] for the detailed proof of this claim.

Definition 18 (Process term skeleton). *Let $K = \langle F, \Pi\!\!R\rangle$ be a component and* p *be a process term of K. We call* skeleton *of* p*, and write* $\mathsf{p}^{\natural}_{F}$*, the constraint set $\bigcup_{i \in I} F_i^{\mathcal{A}}$ where the index set I contains elements i such that $\langle F_i^{\mathcal{A}}, \mathsf{succ}, \tau_i\rangle$ is reachable from $\langle F^{\mathcal{A}}, \langle \mathsf{p}, \bot\rangle\rangle$ using $\rightsquigarrow$.*

Informally, a process term skeleton $\mathsf{p}^{\natural}_{F}$ gathers a set of constraints the validity of which guarantees the secure runs of process p.

4.3 Adequacy

In this section we show that if the skeleton of a process term is compatible with a security policy $\mathcal{SP}$, then the considered process term is secure.

We start by a lemma stating that the security level of actions executed by a process is related to the security level of guards. If the skeleton of a process term is compatible with a security policy then it implies that actions following a guard of privacy level π operate on data with privacy levels higher than or equal to π.

Lemma 3. *Let $K = \langle F_0, \Pi\!\!R\rangle$ be a component with $F_0 = \langle \Sigma, \mathcal{R}_0\rangle$,* $\mathsf{p} = p_0$ *a ground process term of K, $\mathcal{SP}$ a security policy for Σ,* $\mathsf{p}^{\natural}_{F_0}$ *the skeleton of process term p_0 s.t.* $\mathsf{p}^{\natural}_{F_0}$ *is compatible with $\mathcal{SP}$. Consider a transition sequence starting from $\langle F_0, p_0\rangle$: $\langle F_0, p_0\rangle \xrightarrow{\alpha_1} \langle F_1, p_1\rangle \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} \langle F_n, p_n\rangle$ where $\alpha_1 = [g^1 \Rightarrow \mathsf{a}^1, \ldots, \mathsf{a}^1_{n_1}]$, then $\pi(g^1) \sqsubseteq \pi(\alpha_i)$ for all $i \geq 0$.*

Theorem 1. *Let $K = \langle F, \Pi\!\!R\rangle$ be a component,* p *a ground process term of K, $\mathcal{SP}$ a security policy for F,* $\mathsf{p}^{\natural}_{F}$ *the skeleton of process term* p*. If* $\mathsf{p}^{\natural}_{F}$ *is compatible with $\mathcal{SP}$, then,* p *is a secure process term.*

Proof sketch

The proof is done by contradiction. Suppose that p is not secure. The negation of definitions 13 and 14 imply that there are privacy level π, two π-equivalent stores F_1^0, F_2^0 such that $F \cong_\pi F_1 \cong_\pi F_2$, a natural N and two derivations :

$$\langle F_1^0, p^0\rangle \longrightarrow \langle F_1^1, p^1\rangle \longrightarrow \ldots \longrightarrow \langle F_1^N, p^N\rangle$$
$$\langle F_2^0, p^0\rangle \xrightarrow{F_1^0} \langle F_1^1, p^1\rangle \xrightarrow{F_1^1} \ldots \xrightarrow{F_1^{N-1}} \langle F_2^N, p^N\rangle$$

such that $p = p_0$ and for all $j < N$, $F_1^j \cong_\pi F_2^j$ and one of the following points hold: (1) Either $F_1^N \ncong_\pi F_2^N$ (2) or there exists $\langle F^\sharp, p^\sharp\rangle$ such that $\langle F_1^N, p^N\rangle \longrightarrow \langle F_1^{N+1}, p^{N+1}\rangle \longrightarrow^* \langle F^\#, p^\#\rangle$ but $\langle F_2^N, p^N\rangle \not\longrightarrow$ and $F_2^N \ncong_\pi F^\#$.

A contradiction may be derived from both points. For the first point it means that at some point $j \in \{1; \ldots ; n\}$ in the execution path an action has two different effects on two π-equivalent stores F_1^j, F_2^j. But from lemma 1 we can mimic concrete reductions at the abstract level. At this point a simple examination of all elementary action may conclude that the satisfaction of inequations generated by the abstract interpretation is impossible unless a declassifying function is used. Therefore it depends on a value with a privacy level higher than π, thus such a value may be not equal on F_1^j and F_2^j. But in this subcase, and thanks to the definition of $\xrightarrow{F_1^j}$, two different results cannot be computed since they are computed on a common store.

For the second point a contradiction may be derived from lemma 3. if $F^{\#} \not\cong_\pi F_N^2$, then it means that an information of a privacy level *lower or equal* to π has been modified which is impossible since compatibility implies that the level of actions is *higher* than π and thus must not influence π-equivalence between stores.

5. RELATED WORKS

Approaches based on conditional noninterference based on a notion of *downgrader* channels, first presented in [13], share similarities with our work. Intuitively, conditional noninterference disallow high level to interfere with low level unless the interference occurs through a dedicated *downgrading* channel. Of course declassifying functions play the role of downgrading channels in our system. Moreover, some ideas developed in this way of research have their counterparts in our proposition. For instance the fact that the interference relation is intransitive [23, 17] can be interpreted in our setting by our definition of the privacy level of a term. Indeed, this computation does not recursively inspects the structure of a term. It was the case in [7] where the privacy level of a term is computed as the highest privacy level of its subterms. In our system it is no longer the case since a declassifying function may have an argument of privacy level $\top$ but can still be of privacy level $\perp$. So the transitivity of privacy levels on the term tree is broken. Concerning the work of [24], our system extends its results in several directions. First our system may consider infinite computation and second our security policies are more general -they may be defined as complicate as any confluent and terminating rewriting system-.

Nevertheless, our approach is more precise in the sense that in these works, admissible interferences are allowed until the last downgrading action. After there must be no information flows. This paper clarifies our intentions presented in [6]. In our system some of such interferences are allowed if they are initiated by a declassifying function. For instance: $SPY := \xi(PIN); x := SPY + 1$ with ξ a declassifying function, illustrates well this idea. After the use of declassifying function ξ, there may be interferences between high and low levels indirectly coming from that use of ξ. In our system such flows are allowed, since they are controlled by declassifying functions, whereas there are no ways to deal with them in the settings of [23, 17].

Our approach departs from several recent works related to this paper [16, 33, 18] where the same problem is treated. Robust declassification idea is that attackers may not be able to control what information is released. It is not the same approach as ours since we try to take into account situations where the attacker may have such a control (e.g. brute force attacks on password checking) but only using some identified functions. Another difference comes from the fact that the analysis of [18] is defined for simple imperative programs, that is without concurrency while our analysis algorithm can handle concurrency. [16] is closest to our work, it also deals with declassification and how one can make exceptions to the information flow. The underlying formalization is intransitive noninterference and they apply it to an imperative programming-language with threads. In their formalism declassifications are declared to the level of actions, and it is possible to restrict declassifications to certain parts of the security lattice in the security policy. we differ from this approach in several ways. Firstly declassifications are defined at the level of functions, thus providing *security profiles* more elaborate than the definition of a downgrading from one to another level. Secondly, using rewrite rules to express security policies allows one to define more subtleties on security policies. Indeed we can define how to compute the security level of term relatively to the security levels of its subterms. Consider for instance the security level of a list: it may be defined as the join of the security level of its components. These computations over security levels are not possible to express even in polyvariant systems like the one of [21]. Indeed to encode the security policy presented in section 3.2 for the RSA example, it is necessary to compute at the type level. The recent work [15] compares to ours but does not consider concurrent systems. It is also noticeable that our framework copes with the modified strong security condition of [16].

A different approach considers the *quantity* of interferences allowed. This work [20, 19] relies on the definition of a distance, instead of being based on *indistinguishability*, between two processes. It leads to a notion of approximate confinement. Given a description of admissible spies, two agents are approximately confined with respect to some set of spies if there is a distance ϵ such that for all spies the distance between the observable of the attacks of both agents by a spy is smaller than ϵ. We believe this approach is complementary with our one which is more centered on the *quality* of interferences allowed. A drawback of [20, 19] is the limitation of the analysis to finite computations. Therefore it is not easy to see how these ideas may be implemented in our system where process calls may lead to infinite computations. Nevertheless, in [20, 19] is proposed an abstract semantics in order to have a realistic analysis (instead of an exact collecting semantics) giving approximated results. It may be of interest to investigate if, through this abstract semantics, it is possible to adapt those results and combine them with our notion of declared leakages through declassifying functions.

Another interesting point of our proposition is the fact that our notion of declassifying function may also be used to analyze information flows from low to high level. Using our system, it is possible to declare functions $decipher(\perp) \rightarrow \top$. Therefore it is possible to force the result, through an adapted security policy, of functions to be at least at some security level. It allows the possibility to enforce a specific security policy.

6. CONCLUSIONS

In this paper we have defined a notion of security policy in terms of rewrite systems. It is possible to describe the behavior of (in terms of privacy) functions and to declare information leakages (declassifying functions). We have provided a formal definition of these declared interference. Our proposition enriches traditional approaches on security based on non-interference. With these declared interference it is possible to take into account more real-life situations like password verifications, communication of encrypted data through a public channel etc. These situations, to our knowledge, have not been addressed from a noninterference point of view. Moreover the definition of a security policy as a rewriting system. It provides a tool to define very subtle security policies (for instance the privacy level of an encryption mechanism with respect to the status of the used key) is a novelty. Also this formalization is able to take into account the converse approach, for instance the result of deciphering must be secret. It is representable in our system by declaring deciphering functions as functions with security policy $decipher(\bot) \rightarrow \top$. We also have presented an analysis algorithm for this notion of declared interferences.

One key of our proposition is that, in some sense we keep the track of the downgrading/upgrading of information. That is after the use of a declassifying function, the analysis continues, which is not the case in downgrading systems, e.g., [23, 17], where after the last use of a downgrading channel there may be no interference between high and low levels.

Future works include the study of the quantity of high level information leaked to low level (in line with [20]), it could also be declared in the security policy. An orthogonal direction to investigate consists in making security policies more flexible, more precisely it would be a closer study of definition 6 (has the security policy to be terminating) and definition 9 (in what circumstances the right hand side of a rewriting may contain declassified subterms). Another interesting direction to get closer to reality is to take into account a greater precision on security policies. For instance after three unsuccessful password try, the system may prohibit a fourth try. It would imply a deeper use of rewriting rules (real computations, not only rules on privacy levels) in security policies.

7. REFERENCES

[1] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *Proceedings of the* 29th *Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 33 – 44, Portland, Jan. 2002. ACM Press.

[2] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109 – 130, 2002. Special issue: "Merci, Maurice, A mosaic in honour of Maurice Nivat" (P.-L. Curien, Ed.).

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the* 4th *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '77)*, pages 238 – 252, Los Angeles, Jan. 1977. ACM.

[4] N. Dershowitz and D. A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes*, chapter 9, pages 535 –610. Elsevier and MIT Press, Amsterdam, 2001.

[5] R. Echahed and F. Prost. Handling harmless interference (preliminary version). 2003. url: www-leibniz.imag.fr/LesCahiers/Cahier82/ResumCahier82.html.

[6] R. Echahed and F. Prost. Handling declared information leakage (extended abstract). In *Proceedings of Workshop on Issues in the Theory of Security 2005 (WITS'05)*, Long Beach, January 2005.

[7] R. Echahed, F. Prost, and W. Serwe. Statically assuring secrecy for dynamic concurrent processes. 2003. proceedings of PPDP'03, preliminary version avalaible at http://www-leibniz.imag.fr/LesCahiers/2002/Cahier40/Resum-Cahier40.html.

[8] R. Echahed and W. Serwe. Combining mobile processes and declarative programming. In J. Lloyd et al., editors, *Proceedings of the* 1st *International Conference on Computational Logic (CL 2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 300 – 314, London, July 2000. Springer Verlag.

[9] R. Echahed and W. Serwe. Integrating action definitions into concurrent declarative programming. *Electronic Notes in Theoretical Computer Science*, 64, Sept. 2002. special issue: selected papers of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001).

[10] W. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. Springer Verlag, 2000.

[11] R. Giacobazzi and I. Mastroeni. Abstract non-interference. In *Proceedings of the* 31th *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, Venice, Italy, Jan. 2004.

[12] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[13] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

[14] M. Hennessy and J. Riely. Information flow vs. ressource access in the asynchronous pi-calculus. In *Automata, Languages and Programming, 27th International Colloquium, (ICALP'2000), LNCS 1853*, pages 415–427. Springer, 2000.

[15] P. Li and S. Zdancewic. Dowgrading policies and relaxed noninterference. In *Proceedings of the* 32nd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, Long Beach, California, USA, january 2005.

[16] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *2nd ASIAN Symposium on Programming Languages and Systems*, 2004.

[17] J. Mullins. Nondeterministic admissible interference. *Journal of Universal Computer Science*, 6(11):1054–1070, November 2000.

[18] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *17th IEEE Computer Security Foundations Workshop*, pages

172–186, 2004.

[19] A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate confinement under uniform attacks. In M. V. Hermenegildo and G. Puebla, editors, *SAS'02 – Static Analysis, 9th International Symposium*, number 2477 in Lecture Notes in Computer Science, Madrid, Spain, September 2002. Springer.

[20] A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *CSFW'02 – 15th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, 2002.

[21] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, january 2003.

[22] F. Prost. A static calculus of dependencies for the λ-cube. In *Proceedings of the* 15th *Annual IEEE Symposium on Logic in Computer Science (LICS '2000)*, pages 267 – 276, Santa Barbara, 2000. IEEE Computer Society Press.

[23] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference ? In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW-12), Mordano, Italy*, June 1999.

[24] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, Computer Science Laboratory, SRI International, Dec. 1992. Technical Report CSL-92-02.

[25] P. Ryan, J. McLean, J. Millen, and V. Gilgor. Non-interference, who needs it ? In *CSFW'01 – 14th IEEE Computer Security Foundations Workshop*, pages 237 – 238, Cape Breton, Nova Scotia, Canada, June 2001.

[26] P. Ryan and S. Schneider. Process algebra and non-interference. In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[27] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, special issue on Design and Analysis Techniques for Security Assurance*, 2002. to appear.

[28] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. of the 25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 355–364. ACM, 1998.

[29] G. Smith and D. M. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the* 25th *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 355 – 364, San Diego, Jan. 1998.

[30] G. Smith, D. M. Volpano, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167 – 187, 1996.

[31] D. Volpano and G. Smith. Confinements properties for programming languages. *SIGACT News*, 29(3):33–42, 1998.

[32] N. Yoshida and M. Hennessy. Assigning types to processes. In I. C. S. Press, editor, *Proc. of IEEE 15th Ann. Symp. on Logic in Computer Science (LICS'2000)*, pages 334–345, 2000.

[33] S. Zdancewic. A type system for robust declassification. In *Annual Conference on the Mathematical Foundations of Programming Semantics*, 2003.

[34] S. Zdancewic and A. Myers. Robust declassification. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Nova Scotia, Canada, June 2001., 2001.