In ASIAN '03, Lecture Notes in Computer Science, ????:??-??, © Springer, 2003.

Information flow security for XML transformations

Véronique Benzaken[1], Marwan Burelle[1], and Giuseppe Castagna[2]

[1] LRI (CNRS), Université Paris-Sud, Orsay, France
[2] CNRS, Département d'Informatique, École Normale Supérieure, Paris, France

Abstract. *We provide a formal definition of information flows in XML transformations and, more generally, in the presence of type driven computations and describe a sound technique to detect transformations that may leak private or confidential information. We also outline a general framework to check middleware-located information flows.*

1 Introduction

XML is becoming the *de facto* standard document format for data on the Web. Its diffusion however is characterized by two correlated paradoxes:

1. Despite the increasing success of XML for exchanging and manipulating information on the Web, little attention has been paid to characterize and analyze information flows of XML transformations and, specifically, their security implications.
2. As shown by the standardization process, XML documents are intrinsically typed (cf. the notions of well-formedness and validity). Nevertheless, the "standard" programming languages used to manipulate them are essentially untyped, in the sense that even when they are equipped with a type system the latter does not use the type information of XML documents.

If we consider the self-descriptive nature of XML documents, these paradoxes are less surprising than it may seem: XML documents use tags to delimit some content, and these tags can be considered as type information about the content they delimit. Therefore, XML documents—even those that do not contain a DTD—are in some sense "self-typed" constructions and this makes the definition of a type system for XML transformers difficult. As long as a type system often constitutes a first basic step toward the definition of security analyses of transformations, this may partially explain the absence of formal tools to characterize insecure information flows.

Goal. The aim of this article is to define and characterize information flows in XML transformations in order to single out potentially insecure transformations, that is transformations that may leak confidential or private information. To that end we study transformations defined in a typed programming language for XML documents. There are several candidates for such a language, since several attempts have been made in the literature to overcome the second of the XML paradoxes (e.g., HaXML [21], JWIG [3], Xtatic [11], XDuce [13], XQuery [8], and YATL [4]).

In this work we characterize and analyze information flows for transformations defined in ℂDuce [2]. There are several reasons to choose ℂDuce as target language for our study. First and foremost, unlike other XML-oriented languages, ℂDuce is general purpose, in that it provides besides XML types several other datatypes, enabling to program general (even XML unrelated) applications. Second, among the known languages for XML, it possesses the richest type algebra. Finally, its semantic and set theoretic foundations make it a good candidate for defining or hosting a declarative query language (see [2]) and, as such, it nicely fits our scenario of global queries on the Web.

Problems. We said that the difficulty of defining type systems for XML transformations resides in the self-typing nature of the documents. More precisely, this self-typing characteristic induces “type based” (or “type driven”) computations: matching on document tags roughly corresponds to matching documents’ types; similarly, producing elements with different tags corresponds to outputting results of different types. In some sense, typed XML transformations are akin to the application of **typecase** constructions where the different cases may return differently typed results (this is accounted for in ℂDuce by the use of *dynamically bound* overloaded functions). The presence of type driven computations makes the task of capturing information flows much harder and constitutes the main novelty and challenge of this study. In fact we cannot resort to classical data flow analyses since they are usually applied to computational frameworks whose dynamic semantics does not strictly depend on run-time types. A second challenge is that information flows (more precisely, their absence) are usually characterized in terms of the so-called *non-interference* property [12]. Our study demonstrates that in the presence of type driven computations this notion must be modified so as to include static type knowledge, otherwise we end up with very trivial analyses. Finally, the last challenge is to define flow analysis for a pattern-matching based language—as ℂDuce is—since this stands at least two obstacles: (i) pattern matching is a dynamic type-case, therefore we have to propagate type information in the subsequent matches; (ii) the use of a matching policy (first-match as in ℂDuce or best match as in XSLT) induces dependencies among the different components of an alternative pattern as well as among different cases of a pattern-matching expression and this must be taken into account when characterizing information flows (the sole fact of knowing that a pattern did *not* match may produce a flow of information).

Contributions. The contributions of this article are essentially three:

1. it provides a formal definition and study of information flows in the context of XML transformations and, more generally, in the presence of type driven computations;
2. it describes a sound technique to detect XML document transformations that cause insecure information flows, and formally proves its correctness;
3. by defining security annotations and by relating various kind of analyses (static/dynamic, sound/complete) to different query scenarios, it proposes a general framework for checking security of middleware-located information flows.

Example. The development of our presentation can be illustrated by an example. Consider the following XML document which stores names and salaries of the workers of a fictive company. We imagine that while generic users are allowed to perform queries on this document, the information about salaries must only be accessible to authorized users. Therefore we need a way to detect queries that may reveal information about salaries, in order to reject them when they are performed by unauthorized users. A first naive technique to obtain it would be to mark the salary elements and dynamically reject all queries that contain marks in their result. Unfortunately, this approach is clearly inadequate since the information about salaries can be deduced as follows: perform a query that returns the list of all workers whose salary is greater than n and then iterate the query by varying n until we obtain as many different results as workers.

```
<?xml version="1.0"?>
<company>
   <worker>
     <surname>Durand</surname>
       <name>Paul</name>
       <salary>6500</salary>
   </worker>
   <worker>
     <surname>Dupond</surname>
       <name>Jean</name>
       <salary>1800</salary>
   </worker>
   <worker>
       <surname>Martin</surname>
       <name>Jules</name>
       <salary>1200</salary>
   </worker>
</company>
```

A more effective solution is to reject all the queries whose result *accesses* the value of the salary elements. For example consider the following two queries:

[Q1] Get the list of all workers
[Q2] Get the list of all workers whose salary is greater than € 1600

The first query can be always safely executed while the second one must be forbidden to unauthorized users. This can be obtained by enforcing an access control policy. For instance this is done in [7,6] by executing a query on a *view* (in the database sense) obtained by pruning from the XML documents all data the owner of the query has not the right to access.

While enforcing access control is enough for simple policies like the above, it soon becomes inadequate with slightly more complicated policies. For instance imagine that instead of forbidding access to salaries we want to allow queries owned by generic users to access salaries (e.g. for statistical purposes) but in a way that prevents these queries from associating a specific worker with her/his salary. This corresponds to rejecting all queries whose result *depends* both on the value of salary elements and on that of name or surname elements (but queries like Q1 or a query that returns all salaries are acceptable). To enforce this constraint we have to switch from an *access* analysis to a *dependency* (or information flow) analysis.[3]

Causal security policies, such as above, can be formalized by the notion of *non-interference*, that can be restated for XML documents as follows: a set of elements does not interfere with the result of a given query if for all possible contents of the elements the query always returns the same result. In our example, consider the set of all documents obtained from our XML document by replacing the content of salary elements by arbitrary numeric values. Query Q1 is interference-free since when it is applied to all these documents it always returns the same result. Query Q2 instead is not interference-free since its results may differ.

A precise definition of non-interference constitutes the first step of our approach since it defines the set of queries that are safe. The following step is to devise one or more techniques to determine the safety/unsafety of queries. To that end we first classify components that store confidential information by annotating data elements by labels of the form ℓ_t. The ℓ intuitively represents a security classification of the information stored in the element (e.g., public or private, but it could be any label from a possibly unordered set) while t is a type that describes the static information publicly available about the data's content (e.g. for salaries it records that the element stores an integer in a given range)[4]. Next we recast the notion of non-interference in terms of labeled elements, namely, we say that a transformation is free of interference from all elements labeled by ℓ, if its result does not change when the content of the ℓ-labeled elements vary over the type indicated in the label. Our research plan consists of the definition of three different analyses to be used as in the scenario of Figure 1 in the next page. According to it an interactive query (that is, a query that was written to be executed just once) will first pass through a complete static analysis (that rejects transformations that

[3] While for this specific example it is still possible to resort to access control techniques (execute the query on two different views obtained by stripping in one all salaries and in the other all names and surnames) these sole techniques soon become insufficient, as shown by the example in Section 6.

[4] We do not require the existence of any order on such labels (in our framework security policies will be expressed in terms of presence/absence rather than relative order of label), therefore our labels must not to be confounded with the security labels used in multilevel security systems

are manifestly unsafe) and then through a sound dynamic analysis. Instead, programs that are expected to be used several times will pass through a cycle of sound static analysis (possibly preceded by a complete analysis) before being executed without any further dynamic check. In this paper we concentrate on the sound dynamic analysis for $\mathbb{C}$Duce programs, that is, the grayed part of the figure.

Outline. We start in Section 2 by a brief overview of the functional core of $\mathbb{C}$Duce. In Section 3 we formally define the non-interference property for $\mathbb{C}$Duce programs and introduce $\mathbb{C}\text{Duce}_{\mathcal{L}}$ a conservative extension[5] of $\mathbb{C}$Duce in which expressions occurring in a program may appear labeled by security labels. Section 4 is the core of our work. It defines the dynamic analysis that detects interference free programs. The idea is to define an operational semantics for $\mathbb{C}\text{Duce}_{\mathcal{L}}$ such that (i) it preserves the semantics of unlabeled programs and (ii) it ensures that whenever a label ℓ is absent from the final result of a program, then the program is free of interference from all expressions labeled by ℓ. Thanks to these properties the analyzer has simply to label and run a transformation and to refuse to return the (unlabeled) result when this contains unauthorized labels. Of course the heart of the problem is label propagation in pattern matching, whose definition is made difficult by the type driven semantics, the presence of subtyping, and the use of a first-match policy. In Section 5 we prove that our analysis satisfies the aforementioned properties; this goes through proving that $\mathbb{C}\text{Duce}_{\mathcal{L}}$ satisfies the subject-reduction property, that it preserves the semantics of $\mathbb{C}$Duce, and that it constitutes a sound analysis for the non-interference property. The last two points present some technical difficulties (without any practical impact) due to the type system of $\mathbb{C}$Duce that does not satisfy the minimum typing property and induces in $\mathbb{C}\text{Duce}_{\mathcal{L}}$ a non-deterministic semantics: thus the two properties must be proved to hold for *all* possible reductions of a program. In Section 6 we comment a more significant example that illustrates some security policies that cannot be expressed in terms of access control. We conclude our presentation by sketching in Section 7 some research perspectives.

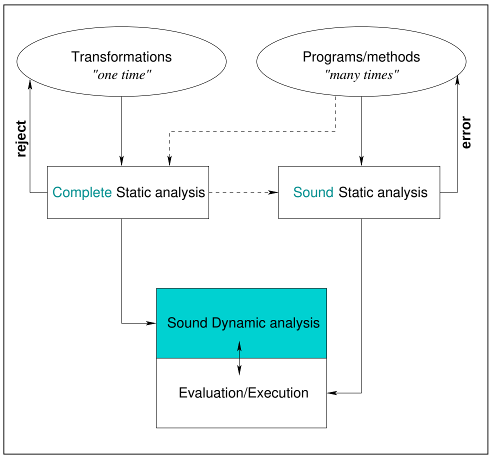

Fig. 1. Analysis scenarios

Related work. Security issues for XML have been addressed by several works but none of them tackles the problem of information flows. They either focus on access control (e.g., [10,7,6]) or on lower level security features such as encryption and digital signatures for which commercial products are becoming available (e.g [14]). For instance, Damiani et al. [7,6] detect accesses to confidential data by applying a static marking of the documents and by dynamically stripping off marked elements. In other words, they deal with access control (confidential *data* is *accessed* for computing the

[5] The extension is conservative with respect to both the type theory and the equational (reduction) theory.

result) whereas our approach accounts for implicit flows (confidential *information* can be *inferred* from the result) like the detection of covert channels. The same holds true for the work of Gabillon and Bruno [10] where access control is performed by running queries on *views* of the XML documents dynamically generated by stripping off unauthorized data. Other works devise flow analyses for programming languages for XML (e.g., [3]) but these analyses are not developed to verify security properties.

The study presented here draws ideas from several sources. The dynamic propagation of labels was first introduced in Abadi et al. [1], where a dependency analysis for call-by-name λ-calculus is defined by extending the reduction semantics to labeled λ-terms. Although their work was not motivated by security reasons (they address optimization issues) what we describe here essentially adapts their technique to type driven reductions. Label propagation was successively used for security purposes in later works, for instance [5,15,16,17]. In particular, in [15] Myers and Liskov use labels for the same purposes as we do; however their security model is defined for and relies on languages that explicitly manipulate labels, while in our or in Abadi's et al. approach, properties are stated for an unlabeled language and labels are introduced on the top of it as a technique to identify (unlabeled) programs satisfying these properties. Finally, all the cited label-based approaches fundamentally differ from the study presented here in that they do not account for type driven semantics (nor for pattern matching) distinctive of XML transformations.

The presence of type driven computations preclude us the use of classical definitions and detection techniques of non-interference (e.g. those of [19,20]), since in this case ignoring static type information would yield a far too weak definition of non-interference. Actually, our notion of non-interference differs from the classical one in that the latter usually relies on a hierarchical structuring of security levels (high-level inputs do not interfere with low-level outputs) while here we spot non-interference of single pieces of code (the value of some data does not interfere with the result of a query). This difference must be understood as the fact that we want to characterize the flows in a single transformation while classical non-interference rather applies to system-wide flows.

2 The CDuce language

ℂDuce is a functional programming language tailored to the manipulation of XML documents, for which it uses its own notation. The XML document in the previous section becomes in ℂDuce the expression on the right. The syntax is mostly self describing: tags are denoted by angle brackets and are followed by sequences. Sequences are delimited by square brackets and in this case are formed by other elements, but in general they may contain expressions of any type (note that some tags are followed by strings as the latter are encoded in ℂDuce as sequences of characters). This expression has the type Company defined right below it. The types of sequences are defined by regular ex-

```
<company>[
  <worker>[
    <surname>"Durand"
    <name>"Paul"
    <salary>[6500] ]
  <worker>[
    <surname>"Dupond"
    <name>"Jean"
    <salary>[1800] ]
  <worker>[
    <surname>"Martin"
    <name>"Jules"
    <salary>[1200] ]
]

type Company = <company>[Worker*]
type Worker  = <worker>[Sname Name Salary]
type Sname   = <surname>String
type Name    = <name>String
type Salary  = <salary>[Int]
```

pressions on types. For example, the first type declaration states that a company is a sequence tagged by <company> and composed of zero or more worker elements. Had we defined the type of workers as follows:

```
Worker2 = <worker ceo=?Bool>[ Sname Name Salary (Email | Tel)? ]
```

then workers elements would have an optional (as indicated by =?) boolean attribute and list a last optional element that is either of type Tel or of type Email. Note also that Worker is a subtype of Worker2 since every value of the former type is also a value of the latter type. The queries defined in the previous section can be expressed as:

```
[Q1:] let <company>x = mycompany in transform x with <worker>[ y z _ ] → [ <worker>[ y z ] ]

[Q2:] type MoreThanMe = <salary>[1600--*]
      let <company>x = mycompany in transform x with <worker>[ y z MoreThanMe ] → [ <worker>[ y z ] ]
```

where mycompany is a variable that denotes our previous <company> XML document.

In the first query the let expression matches mycompany against a pattern that binds the variable x to the sequence of worker elements of mycompany. The transform expression applies to each element of a sequence a pattern that returns a sequence and then concatenates all the results (non matching elements are simply discarded). In this case it transforms the sequence of Worker elements into a sequence of <worker>-tagged elements containing just surname and name elements (y and z respectively capture the surname and name elements, while the _ pattern matches every value). Had we defined the type Company as <company>[(Worker|Client)*], that is a heterogeneous sequence of workers and clients, then Q1 would still have returned the same result as transform would discard client elements. The query Q2 is similar, with the only difference that the transform pattern matches only if the salary element has type MoreThanMe, that is, if its content is an integer greater than or equal to 1600.[6]

A detailed presentation of ℂDuce is out of the scope of this paper. The interested reader can refer to [2] and try the interactive prototype at www.cduce.org. For the purpose of this work it will suffice to say that all ℂDuce constructions are encoded in the following core language defined in [9]:

$$t ::= \alpha \mid C \mid \mu\alpha.C \quad C ::= A \mid \neg C \mid C\vee C \mid C\wedge C \quad A ::= t \to t \mid t\times t \mid b \mid \mathbf{0} \mid \mathbf{1}$$

We use t to range over types, C over boolean combinations, A over atoms, and b over basic types (Int, Char, ...); $\mathbf{0}$ and $\mathbf{1}$ denote respectively the empty and top type, and together with arrows and products they form the atoms of arbitrary boolean combinations (negation, union, and intersection types) and recursive types $\mu\alpha.C$.[7] The expressions of the language are:

$$e ::= c \mid x \mid \ \mathsf{fun}\ f^{(t_1\to s_1;\dots;t_n\to s_n)}(x).e \mid e_1e_2 \mid (e_1,e_2) \mid \ \mathsf{match}\ e\ \mathsf{with}\ p_1\to e_1 \mid p_2\to e_2$$

The syntax is essentially that of a higher-order functional language with constants, pairs and pattern-matching. What distinguishes it from similar languages is the definition of patterns (given later) and of functions. In the latter note that the name of a function (which may appear in its body as functions can be recursive) is annotated by a non empty list of arrow types (this list is called the *interface* of the function). The presence

---

6 The results the two queries are the central and right expressions in Figure 4 from which we erase all the label annotations of the form ℓ_t:, m_t:, or n_t:

7 The distinction between atoms and combinations is introduced in order to avoid meaningless recursive types such as $\mu\alpha.\alpha \vee \alpha$

of more than one arrow type declares the overloaded nature of the function, which has all these types, thus their intersection as well (see the typing rule for functions later on).

The operational semantics is defined in terms of the set $\mathcal{V}$ of *values*, ranged over by v and formed by all *closed* and *well-typed* ℂDuce expressions of the form

$$v \quad ::= \quad c \mid \mathsf{fun}\, f^{(t_1 \to s_1; \ldots; t_n \to s_n)}(x).e \mid (v_1,v_2)$$

The semantics is given by the reduction relation $\rightsquigarrow$, which is defined as follows:

$$\begin{array}{rll}
(v_1 v_2) \rightsquigarrow e[v_1/f; v_2/x] & & (v_1 = \mathsf{fun} f^{(\ldots)}(x).e) \\
(\,\mathsf{match}\ v\ \ \mathsf{with}\ p_1 \to e_1 \mid p_2 \to e_2) \rightsquigarrow e_1[v/p_1] & & (v/p_1 \neq \Omega) \\
(\,\mathsf{match}\ v\ \ \mathsf{with}\ p_1 \to e_1 \mid p_2 \to e_2) \rightsquigarrow e_2[v/p_2] & & (v/p_1 = \Omega\ ,\ v/p_2 \neq \Omega)
\end{array}$$

where Ω denotes the matching failure and $e[v/p]$ is the expression obtained by applying the substitution v/p to e. The first rule is standard β-reduction for recursive functions. The second rule states that if the first pattern matches, then the expression of the first branch is executed after having applied to it the substitution v/p_1 of the variables captured by p_1 when matching v. The second rule states that the same happens for the second branch when the first pattern fails and the second matches (the static type system of ℂDuce ensures that the patterns cannot both fail).

Reductions can take place in any *evaluation context*. A *context*, denoted by $\mathcal{C}$ is an expression with a hole substituted for a sub-expression. An *evaluation context*, denoted by $\mathcal{E}$, is a context whose hole is not in the scope of an abstraction or of a pattern and which induces a (leftmost) evaluation order on applications and pairs. Formally, $e \rightsquigarrow e'$ implies $\mathcal{E}[e] \rightsquigarrow \mathcal{E}[e']$, where $\mathcal{E}$ is defined as

$$\mathcal{E} ::= [\,] \mid (\mathcal{E},e) \mid (v,\mathcal{E}) \mid \mathcal{E}e \mid v\mathcal{E} \mid \ \mathsf{match}\ \mathcal{E}\ \ \mathsf{with}\ p_1 \to e_1 \mid p_2 \to e_2$$

As anticipated, key definitions for ℂDuce are those of pattern and pattern matching. A regular tree built from the following grammar

$$p ::= x \mid t \mid p_1 \& p_2 \mid p_1 | p_2 \mid (p_1,p_2) \mid (x := c)$$

is a pattern if (i) on every infinite branch of it there are infinite occurrences of the pair constructor[8], (ii) patterns in an alternative capture the same variables, and (iii) patterns in a conjunction capture pairwise distinct variables. The semantics of patterns is defined by the matching operation: the matching of value v against a pattern p is denoted by v/p and returns either Ω (matching failure) or a substitution from the variables of p into $\mathcal{V}$:

$$\begin{array}{llll}
v/t = \{\} & \text{if } v : t & v/t = \Omega & \text{if } v : \neg t \\
v/x = \{x \mapsto v\} & & v/(x := c) = \{x \mapsto c\} & \\
v/p_1|p_2 = v/p_1 & \text{if } v/p_1 \neq \Omega & v/p_1|p_2 = v/p_2 & \text{if } v/p_1 = \Omega \\
v/p_1\&p_2 = v/p_1 \otimes v/p_2 & & v/(p_1,p_2) = \Omega & \text{if } v \text{ is not a pair} \\
(v_1,v_2)/(p_1,p_2) = v_1/p_1 \otimes v_2/p_2 & & &
\end{array}$$

where $\gamma_1 \otimes \gamma_2$ is Ω when $\gamma_1 = \Omega$ or $\gamma_2 = \Omega$ and otherwise is the substitution $\gamma \in \mathcal{V}^{\mathit{Dom}(\gamma_1) \cup \mathit{Dom}(\gamma_2)}$ defined as $\gamma(x) = \gamma_1(x)$ when $x \in \mathit{Dom}(\gamma_1) \backslash \mathit{Dom}(\gamma_2)$, as $\gamma(x) = \gamma_2(x)$ when $x \in \mathit{Dom}(\gamma_2) \backslash \mathit{Dom}(\gamma_1)$, and as $\gamma(x) = (\gamma_1(x),\gamma_2(x))$ when $x \in \mathit{Dom}(\gamma_1) \cap \mathit{Dom}(\gamma_2)$.

The semantics of patterns is rather intuitive. There are two possible causes of failure for a pattern matching: a type constraint which is not satisfied by the matched

[8] Infinite trees represent recursive patterns while the condition on infinite branches endows the set of patterns with a well-founded order that ensures termination for the algorithms of typing and matching.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (\textit{var}) \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1,e_2) : t_1 \times t_2}\ (\textit{pair}) \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 e_2 : t_1 \bullet t_2}\ (\textit{appl})$$

$$\frac{\Gamma \vdash e : s \leq \wr p_1 \int \vee \wr p_2 \int \quad \Gamma, (s_i/p_i) \vdash e_i : t_i}{\Gamma \vdash \ \mathsf{match}\ e\ \ \mathsf{with}\ p_1 \rightarrow e_1 \,|\, p_2 \rightarrow e_2 : \bigvee_{\{i | s_i \not\simeq \mathbf{0}\}} t_i}\ (\textit{match}) \quad (\text{for } s_1 \equiv s \wedge \wr p_1 \int,\ s_2 \equiv s \wedge \neg \wr p_1 \int)$$

$$\frac{(\forall i) \quad \Gamma, (x : t_i), (f : \bigwedge_{i=1..n} t_i \rightarrow s_i) \vdash e : s_i}{\Gamma \vdash \ \mathsf{fun}\ f^{(t_1 \rightarrow s_1; \dots; t_n \rightarrow s_n)}(x).e : \bigwedge_{i=1..n} t_i \rightarrow s_i}\ (\textit{abstr}) \qquad \frac{\Gamma \vdash e : s \leq t}{\Gamma \vdash e : t}\ (\textit{subsum})$$

Fig. 2. Typing rules

value, or a pair pattern applied to a value that is not a pair. The alternative pattern $p_1|p_2$ has a first-match policy: the object is matched against p_2 if and only if the matching against p_1 fails. When a variable x appears on both sides of a pair pattern, the two captured elements are paired together as expressed by the third case of the definition of $\otimes$. The default-value pattern $(x := c)$ usually appears in the right-hand side of an alternative pattern to return a default value when the left-hand side fails. The combination of multiply-occurring variables and default-value patterns within recursive patterns allow very expressive captures: for instance, the recursive pattern $p = (x\&\,\mathsf{Int},p)|(\_,p)|(x := \mathsf{nil})$ binds x to the sublist of all integers occurring in a heterogeneous list (encoded *à la* lisp) when this is matched against it. On the whole, all a pattern does is to decompose values in subcomponents that are bound to variables and/or matched against types, whence the type driven computation.

Values are also at the basis of the definition of subtyping. Indeed, the key characteristic of ℂDuce (from which all others derive) is that the subtyping relation is defined semantically via a set theoretic interpretation of the types. This interpretation is very easy to define: a type is interpreted as the set of all values that have that type. So for example $t_1 \times t_2$ is the set of all expressions (v_1,v_2) where v_i is a value of type t_i; $t_1 \rightarrow t_2$ is the set of all closed functional expressions $\mathsf{fun}\, f^{(s_1;\dots;s_n)}(x)e$ that when applied to a value of type t_1 return a result in t_2; union, intersection, and negation of types have the usual set theoretic meaning. This (semantic) interpretation is used to define the (syntactic) subtyping relation: a type s is a subtype of t if the interpretation of s is contained in the interpretation of t.

Typing rules for ℂDuce are summarized in Figure 2. For a more detailed explanation the reader can refer to [9]. Rules (*var*), (*pair*), and (*subsum*) are standard. The rule (*appl*) is less common, since one usually expects t_1 to be of the form $s_1 \rightarrow s_2$ and the rule to infer for the application the type s_2 provided that $t_2 \leq s_1$. However because of the presence of boolean combinations t could be for example of the form $((s_1 \rightarrow u_1) \wedge (s_2 \rightarrow u_2))$. To that end we introduce a partial binary operator on types $\bullet$. Intuitively $t \bullet s$ denotes the least type (if it exists) such that $t \leq s \rightarrow (t \bullet s)$ and the rule (*appl*) does not fail only if the operator is defined (in practice, we subsume t_1 to the least arrow type with domain t_2). So, for example, if our function of type $((s_1 \rightarrow u_1) \wedge (s_2 \rightarrow u_2))$ is applied to an expression of type $(s_1 \vee s_2)$, then the type system will infer that the result of the application has type $((s_1 \rightarrow u_1) \wedge (s_2 \rightarrow u_2)) \bullet (s_1 \vee s_2) = (u_1 \vee u_2)$. Similarly we introduce two projections operators π_1 and π_2 such that $\pi_1(t)$ (respectively $\pi_2(t)$) is the least type that makes $t \leq \pi_1(t) \times \mathbf{1}$ (respectively $\mathbf{1} \times \pi_2(t)$) hold. It is important to note that $\bullet$, π_1, and π_2 are all computable (once more see [9]).

Rule (*abstr*) is not very standard, either. It states that a (possibly overloaded) function is typed by the intersection of all the types declared in its interface: we repeat the check for each type in the interface and handle the typing of recursive calls by recording for f the expected type. Actually the rule in Figure 2, is a simplification of the $\mathbb{C}$Duce rule in [9], the latter is very technical and makes the minimum typing property fail (so that there does not exist a canonical type representing the set of all types of a given expression). However, we will see that this does not affect our analysis and, therefore, that the rule defined in Figure 2 is enough.

Finally the rule (*match*) states that the type of a matching operation is the union of the types of the branches[9], and for that it uses several auxiliary notations: $\equiv$ denotes syntactic equivalence of types whereas $\simeq$ denotes the semantic one ($s \simeq t$ only if s and t denote the same set of values); s/p denotes the type environment that assigns types to the capture variables of p when this is matched against a value of type s; $\wr p \int$ denotes the *type* accepted by p, that is, the type whose interpretation is the set of all values for which p does not fail: $\wr p \int = \{v \mid v/p \neq \Omega\}$. Again, $\wr\ \int$ is computable. The type system is sound since it satisfies the subject reduction property: if $\Gamma \vdash e : t$ and $e \leadsto^* e'$, then $\Gamma \vdash e' : t$ (where $\leadsto^*$ means "reduces in zero or more steps").

3 Non-interference and labels

The first step toward the definition of our security analysis is the characterization of information flows. In general, these are difficult to define and therefore it is customary to rather characterize their absence via the *non-interference* property: given an expression e, a sub-expression e' occurring in e does not interfere with e if and only if whenever e returns some result, then also every expression obtained from e by replacing e' by a different expression yields the same result. Note that this (informal) definition does not involve types and because of that it is unsuitable to describe non-interference for type driven semantics. A definition of non-interference for $\mathbb{C}$Duce transformations must take types into account.

Types have been widely used in previous work on language-based information-flow security (see [18] for a very broad review) and non-interference definitions have been given for typed languages. What is new in our framework is that the essence of the definition of non-interference relies on types. In particular, changing the type of an expression occurrence in a program can change the non-interference properties of that expression occurrence. Therefore our definition of non-interference is stated *with respect to* a type t:

Definition 1 (Occurrence). *Let* e *be a* $\mathbb{C}$*Duce expression. An* occurrence Δ *of* e *is a (root starting) path of the abstract syntax tree of* e*. If* Δ *is an occurrence of* e*, we use* e_Δ *to denote the sub-expression of* e *occurring at* Δ*, and* $\mathcal{C}^e_\Delta[\,]$ *to denote the context obtained from* e *by inserting a hole at* Δ*. Thus* $e = \mathcal{C}^e_\Delta[e_\Delta]$.

Definition 2 (Non-interference w.r.t. t). *Let* e *be a* $\mathbb{C}$*Duce expression,* Δ *an occurrence of* e*, and* t *a type. The occurrence* Δ does not interfere in e with respect to t *if and only if for all* $\mathbb{C}$*Duce values* v *and* v'*, if* $\varnothing \vdash v' {:} t$ *and* $e \leadsto^* v$ *then* $\mathcal{C}^e_\Delta[v'] \leadsto^* v$.

Let e be the application of a query to an XML document that stores some information of type t in an element located at Δ. We want to define when the query allows one to

[9] Precisely, of the branches that have a chance to match, that is those for which $s_i \not\simeq \mathbf{0}$. This distinction matters when typing overloaded functions: see [9].

infer information about Δ more precise than its type. The definition above states that the information stored in Δ is not disclosed if the fact of storing any value v' of type t in Δ does not change the result of the query.

The definition is reasonable as it simply encompasses the static knowledge about the type of an occurrence. For instance, on our example it states that a query is safe (i.e. interference free) if and only if it cannot distinguish two documents that contain two different *integers* in corresponding salary elements. Similarly, a query that distinguishes documents with integer salaries from documents in which salaries elements contain boolean values is safe, too[10]: this is reasonable to the extent that such a test cannot in practice be performed as it would contradict the static knowledge we (or an attacker) have of the document. This new definition induces also a very nice interpretation of security, according to which *a transformation is safe if it does not reveal (about confidential data) any information that is not already statically known*.

A last more technical point. In Definition 2 we tested non-interferent occurrences against values rather than against generic expressions. This simplifies the definition inasmuch as we do not have to take into account the typing and the possible capture of variables that are free in the expressions inserted in the context. However, this does not undermine the generality of our definition as shown by Proposition 1:

Definition 3 ((Γ/Θ)-contexts). *Let Γ and Θ be type environments, t and t' types, and $\mathcal{C}[\ ]$ a context. $\mathcal{C}[\ ]$ is a $(\Gamma/\Theta, t/t')$-context if $\Gamma \vdash \mathcal{C}[] : t$ is provable under the hypothesis that $\Theta' \vdash []{:}t'$ holds for every extension Θ' of Θ.*

Proposition 1 (Generality). *Let e be a $\mathbb{C}$Duce expression such that $\Gamma \vdash e : t$. Let Δ be an occurrence that does not interfere in e w.r.t. t'. Then for all Γ', Γ'', and e', if $\Gamma', \Gamma, \Gamma'' \vdash e'{:}t'$, $e \rightsquigarrow^* v$, and $\mathcal{C}^e_\Delta[\,]$ is a $((\Gamma',\Gamma)/\Gamma'', t/t')$-context, then $\mathcal{C}^e_\Delta[e'] \rightsquigarrow^* v$.*

Our next step consists in defining a way to identify occurrences. This can be easily obtained by marking $\mathbb{C}$Duce expressions by security labels. Thus we define $\mathbb{C}\text{Duce}_\mathcal{L}$ obtained from $\mathbb{C}$Duce by adding expressions of the form "$\ell_t : e$", where t is a type and ℓ is a metavariable ranging over a set $\mathcal{L}$ of labels. As anticipated labels are indexed by types that record the static knowledge of the expression at issue. The type is used to verify non-interference of the expression. This is considered not to interfere with a given computation if the fact of making the labeled occurrence vary over the values of the type specified by the label does not affect the final result of the computation.

Of course, this is sensible only if the fact of making the occurrence vary over such a type yields well-typed expressions. In other terms, as suggested by Proposition 1, the type indexing a label must be a viable type for the expression marked by the label. In order to formalize this property we endow $\mathbb{C}\text{Duce}_\mathcal{L}$ with a type system formed by all the typing rules of $\mathbb{C}$Duce, summarized in Figure 2, plus the one on right.

$$\frac{\Gamma \vdash e : s \qquad s \leq t}{\Gamma \vdash (\ell_t{:}e) : t}$$

By definition a $\mathbb{C}\text{Duce}_\mathcal{L}$ expression is a $\mathbb{C}$Duce expression where some subexpressions are marked by a list of labels. We call *strip* the function that transforms the former into the latter by erasing all the lists of labels and we denote if by $\lfloor\ \rfloor$.

Technically, we consider the syntax of $\mathbb{C}\text{Duce}_\mathcal{L}$ as a description of a decoration of the syntax tree of e in the sense that the (lists of) labels are not nodes of the syntax

[10] More interestingly, had we defined salaries to be of type <salary>[1600- -*], then also the query Q2 would result safe (i.e. interference free).

trees but tags marking these nodes.[11] The reason for this choice is that in this way the same path Δ denotes an occurrence of a $\mathbb{C}$Duce$_{\mathcal{L}}$ expression and the corresponding occurrence in the stripped expression. In this way we avoid to define complex mappings between occurrences of expressions in $\mathbb{C}$Duce$_{\mathcal{L}}$ and the corresponding ones in the stripped version. In particular this makes the following property hold

Proposition 2. *Let e be a $\mathbb{C}$Duce$_{\mathcal{L}}$ term with an occurrence Δ, then (i) $\lfloor e_\Delta \rfloor = \lfloor e \rfloor_\Delta$ and (ii) $\lfloor \mathcal{C}^e_\Delta [e_\Delta] \rfloor = \mathcal{C}^{\lfloor e \rfloor}_\Delta [\lfloor e \rfloor_\Delta]$*

which yields a simpler statement of the non-interference theorem (Theorem 2).

4 Security analysis

Our analysis is defined by endowing $\mathbb{C}$Duce$_{\mathcal{L}}$ with an operational semantics defined so that (i) it preserves $\mathbb{C}$Duce semantics on stripped terms ($e \rightsquigarrow^* v \Rightarrow \lfloor e \rfloor \rightsquigarrow^* \lfloor v \rfloor$) and (ii) label propagation provides a sound non-interference analysis for $\mathbb{C}$Duce expression (i.e., if a label is not propagated, then the labeled expression does not interfere with the result). It is not difficult to define a trivially sound analysis (just propagate all labels)[12]. The hard task is to define a fine-grained analysis that propagates as few labels as possible. Needless to say that the core of the definition is label propagation in pattern matching, where several kinds of information flows are possible:

- Direct (or "explicit") information flows, due to the binding of labeled expressions to variables, such as in: match e with <worker>[x y <salary>**[z]**] ➔ ...**z**... | ... where the value of the salary is bound to **z** and flows through it into the first branch expression.
- Indirect (or "implicit") information flows due to patterns: the information flows by the deconstruction of the matched value and/or the satisfaction of type constraints, such as in: match e with <worker>[x y **<salary>[1--900]**] ➔ **e₁** | ...
  where the information of the range of the salary flows into **e₁**.
- Indirect information flows due to use of a first match policy: information can be acquired from the failure of the previous branch:
  match e with <worker>[x y **<salary>[1--900]**] ➔e₁ | _ ➔**e₂**
  where the information that the salary value is *not* in (1--900) flows into **e₂**.

The last example also shows that non-interference is undecidable as the non-interference of the value of the salary element is equivalent to deciding the equivalence of e₁ and e₂ ($\mathbb{C}$Duce is Turing-complete).

Once we have established whether a label must be propagated or not, here comes the problem to decide with which type we have to decorate it so as not to infringe subject-reduction. Consider the example of the application of a labeled function value to another value: $(\ell_{s\to t}{:}v_1)v_2$. Surely the function, seen as a piece of data, interferes with the computation. Therefore its label must be propagated. A sound solution is to reduce the application to $\ell_u{:}(v_1 v_2)$, for a suitable choice of u. Here the choice for u is easy: the type system ensures that v_2 is of type s therefore it suffices to take $u = t$. When the type of the label is not an arrow but a generic type t', then we resort to the $\bullet$ operator and set u equal to $t' \bullet s'$ for some s' such that $v_2 : s'$.

[11] More formally we consider that a $\mathbb{C}$Duce$_{\mathcal{L}}$ term denotes a pair formed by a $\mathbb{C}$Duce term and a map from the (root starting) paths of the abstract syntax tree of the term to possibly empty lists of labels.

[12] The analogous trivially complete analysis is the one that does not propagate any label

$$
\begin{array}{ll}
v /\!\!/ x \;=\; \varnothing_{\mathbf{ok}} & \\
v /\!\!/ (x := c) \;=\; \varnothing_{\mathbf{ok}} & \\
c /\!\!/ p \;=\; \varnothing_{\mathbf{ok}} & \text{if } c \in \llparenthesis p \rrparenthesis \\
c /\!\!/ p \;=\; \varnothing_{\mathbf{fail}} & \text{if } c \notin \llparenthesis p \rrparenthesis \\
(\ell_t : v) /\!\!/ p \;=\; \varnothing_{\mathbf{fail}} & \text{if } t \wedge \llparenthesis p \rrparenthesis \simeq \mathbf{0} \\
(\ell_{t'} : v) /\!\!/ t \;=\; \varnothing_{\mathbf{ok}} & \text{if } (t' \leq t) \\
(\ell_{t'} : v) /\!\!/ t \;=\; \ell :: (v /\!\!/ t) & \text{if } (t \wedge t' \not\simeq \mathbf{0}) \wedge (t' \not\leq t) \\
(v_1, v_2) /\!\!/ t \;=\; (v_1 /\!\!/ \pi_1(t))@(v_2 /\!\!/ \pi_2(t)) & \text{if } \pi_i(t) \text{ are defined and } v_i /\!\!/ \pi_i(t) \neq \varnothing_{\mathbf{fail}} \\
(v_1, v_2) /\!\!/ t \;=\; \varnothing_{\mathbf{fail}} & \text{otherwise} \\
(\mathsf{fun}\ f^{(t_1;\dots;t_n)}(x).e) /\!\!/ t \;=\; \varnothing_{\mathbf{ok}} & \text{if } \mathsf{fun}\ f^{(t_1;\dots;t_n)}(x).e/t \neq \Omega \\
(\mathsf{fun}\ f^{(t_1;\dots;t_n)}(x).e) /\!\!/ t \;=\; \varnothing_{\mathbf{fail}} & \text{if } \mathsf{fun}\ f^{(t_1;\dots;t_n)}(x).e/t = \Omega \\
v /\!\!/ p_1 \& p_2 \;=\; \varnothing_{\mathbf{fail}} & \text{if } (v_1 /\!\!/ p_1) = \varnothing_{\mathbf{fail}} \text{ or } (v_2 /\!\!/ p_2) = \varnothing_{\mathbf{fail}} \\
v /\!\!/ p_1 \& p_2 \;=\; (v /\!\!/ p_1)@(v /\!\!/ p_2) & \text{otherwise} \\
(\ell_t : v) /\!\!/ (p_1,p_2) \;=\; \ell :: (v /\!\!/ (p_1,p_2)) & \text{if } (t \wedge \llparenthesis (p_1,p_2) \rrparenthesis \not\simeq \mathbf{0}) \\
(v_1, v_2) /\!\!/ (p_1,p_2) \;=\; \varnothing_{\mathbf{fail}} & \text{if } (v_1 /\!\!/ p_1) = \varnothing_{\mathbf{fail}} \text{ or } (v_2 /\!\!/ p_2) = \varnothing_{\mathbf{fail}} \\
(v_1, v_2) /\!\!/ (p_1,p_2) \;=\; (v_1 /\!\!/ p_1)@(v_2 /\!\!/ p_2) & \text{otherwise} \\
(\mathsf{fun}\ f^{(t_1;\dots;t_n)}(x).e) /\!\!/ (p_1,p_2) \;=\; \varnothing_{\mathbf{fail}} & \\
(\ell_t : v) /\!\!/ p_1|p_2 \;=\; ((\ell_t : v) /\!\!/ p_1) & \text{if } t \leq \llparenthesis p_1 \rrparenthesis \\
(\ell_t : v) /\!\!/ p_1|p_2 \;=\; ((\ell_t : v) /\!\!/ p_2) & \text{if } t \wedge \llparenthesis p_1 \rrparenthesis \simeq \mathbf{0} \\
(\ell_t : v) /\!\!/ p_1|p_2 \;=\; \ell :: (v /\!\!/ p_1|p_2) & \text{otherwise} \\
v /\!\!/ p_1|p_2 \;=\; \varnothing_{\mathbf{ok}} & \text{if } v /\!\!/ p_1 = \varnothing_{\mathbf{ok}} \\
v /\!\!/ p_1|p_2 \;=\; \varnothing_{\mathbf{fail}} & \text{if } v \neq c, v \neq \ell_t{:}v' \text{ and} (v_1 /\!\!/ p_1) = (v_2 /\!\!/ p_2) = \varnothing_{\mathbf{fail}} \\
v /\!\!/ p_1|p_2 \;=\; (v /\!\!/ p_1)@(v /\!\!/ p_2) & \text{if } v \neq c, v \neq \ell_t{:}v', v /\!\!/ p_1 \neq \varnothing_{\mathbf{ok}}, \text{ and} \\
 & \quad (v /\!\!/ p_1) \text{ and } (v /\!\!/ p_2) \text{ are not both equal to } \varnothing_{\mathbf{fail}}
\end{array}
$$

Fig. 3. Propagating labels with patterns

Formally, we start by (i) defining $\mathbb{C}\mathrm{Duce}_{\mathcal{L}}$ values, which are obtained by adding to the definition of $\mathbb{C}$Duce values the production $v ::= \ell_t : v$; (ii) defining the semantics of v/t for the new values in the following case: $(\ell_t : (v_1, v_2))/(p_1,p_2) = v_1/p_1 \otimes v_2/p_2$; and (iii) adding to evaluation contexts the production $\mathcal{E} ::= \ell_t : \mathcal{E}$. Note however that we do not define a pattern for labels, as we want labels to describe information flow, rather than to affect it.

Next we define the operator "$/\!\!/$" that calculates the set of labels that must be propagated at pattern matching. More precisely, $e /\!\!/ p$ denotes the list of all labels that must be propagated when matching expression e against pattern p, and is defined in Figure 3 (where "@" and "::" are the "append" and "cons" list operators, respectively). This definition forms the core of our analysis, therefore it is worth explaining it in some details. Let $\mathcal{L}(e)$ denote the set of labels occurring in e, then all the labels in $v /\!\!/ p$ are contained in $\mathcal{L}(v)$. The idea is to collect in $v /\!\!/ p$ all the labels that mark expressions whose value *may* affect the result of the matching. Intuitively, the definition of "$/\!\!/$" must ensure that if a label ℓ in $\mathcal{L}(v)$ is not in $v /\!\!/ p$, then for all v' obtained by substituting in v some ℓ-labeled occurrences by "admissible" (i.e. that respect the type constraints indexing ℓ) values, the match v'/p either always succeeds or always fails.[13]

The first two cases in Figure 3 are the easiest ones insofar as variables and default-value patterns always succeed; therefore no label needs to be propagated.

Matching a constant is also simple as there is no label to propagate. Note, however, that we use an index to distinguish two different cases of non-propagation: the case in which labels are not propagated because the pattern always fails ($\varnothing_{\mathbf{fail}}$) and the case

[13] However, this is not enough to ensure non-interference: for instance consider $(\ell_{\mathsf{Bool}} : v) /\!\!/ (\mathsf{true}\&(x := 1))|(\mathsf{false}\&(x := 0))$ which always succeeds.

in which they are not propagated because the pattern always succeed ($\varnothing_{\mathbf{ok}}$). When we match a labeled value against a pattern p and the type of the label does not intersect the type accepted by p, then we know that making v varying over t always fails.

The case for a type constraint pattern is the key one, as $\mathbb{C}$Duce's pattern matching is nothing but a highly sophisticated type-case with capture variables: if $t' \wedge t \simeq \mathbf{0}$, then we are in the previous case so v/t always fails; if $t' \leq t$, then for every v of type t', v/t always succeeds so no label is propagated; otherwise, for all v' in the intersection of t and t' v'/t succeeds, while for v' in their difference it fails, thus we propagate ℓ and check for other labels.

When the matched value is a pair then we propagate the union of the labels propagated by matching each sub-value against the corresponding projection of t provided that: (i) the projections are defined, because otherwise we are matching a pair against a type with no product among its super-types (e.g., $s \rightarrow t$) and the pattern always fails, and (ii) none of the two sub-matching fails, because the whole pattern would then fail and therefore it would be silly to propagate the labels of the other component. Note that this case uses the distinction between $\varnothing_{\mathbf{ok}}$ and $\varnothing_{\mathbf{fail}}$, the other cases being in conjunction and pair patterns—which fail if any sub-check fails—and in disjunction patterns—which fail if both sub-checks fail—. Since there is no pattern that deconstructs functions, these can be soundly matched only against types (or captured). Note however that the type of a function value is fixed by its interface. Therefore even if we make labeled expressions occurring in the function vary, this does not affect the type of the function, ergo the result of pattern matching. For this reason $(\mathsf{fun}\ f^{(t_1,\ldots,t_n)}(x).e) /\!\!/ t$ is dealt in the same way as constant values. The cases of conjunction patterns as well as those for pair patterns need no particular insight. It may be worth just noticing that in the first case of pair patterns ℓ is propagated also when the pattern always succeeds (i.e. $t \leq \wr(p_1,p_2)\int$) as the simple fact of deconstructing the pair may yield interference[14].

The cases for alternative patterns are more interesting as they take into account the use of the first match policy. In particular in the first two equations for match, propagation is calculated only on p_1 or p_2 according to whether the first match always succeeds or always fails. If instead the result cannot be determined with certainty, then the label is propagated and the search for other labels is continued on v. All remaining rules are straightforward.

Now that we have determined the labels to propagate we have to choose the type constraints to decorate them with. Let L be a list $[\ell^1\,\ell^2 \ldots \ell^n]$ of labels, t a type, and e an expression, we use the notation $(L)_t{:}e$ to denote the expression $\ell^1_t : \ell^2_t : \ldots \ell^n_t : e$ obtained by prefixing e by the t-indexed labels in L. This notation is used to index the labels deduced by "$/\!\!/$" and define the operational semantics of $\mathbb{C}\mathrm{Duce}_{\mathcal{L}}$ as follows (a better definition can be found in the on-line extended version available at www.cduce.org):

$v_1 v_2 \rightsquigarrow e[v_1/f; v_2/x]$	if $v_1 = \mathsf{fun} f^{(\ldots)}(x).e$
$(\ell_t : v_1)v_2 \rightsquigarrow \ell_{t \bullet s} : (v_1 v_2)$	if $\varnothing \vdash v_2 : s$ and $t \bullet s$ is defined
$\mathsf{match}\ v\ \mathsf{with}\ p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \rightsquigarrow (v /\!\!/ p_1)_t : e_1[v/p_1]$	if $v/p_1 \neq \Omega$ and $\varnothing \vdash e_1[v/p_1] : t$
$\mathsf{match}\ v\ \mathsf{with}\ p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \rightsquigarrow (v /\!\!/ \wr p_1 \int)_t : (v /\!\!/ p_2)_t : e_2[v/p_2]$	if $v/p_1 = \Omega, v/p_2 \neq \Omega$ and $\varnothing \vdash e_2[v/p_2] : t$

The rule for "unlabeled" applications does not change, while the one for applications

[14] For example , consider $(\ell_{\mathsf{Bool} \times t} : v) /\!\!/ (\ (\mathsf{true} \& (x := 1)) | (\mathsf{false} \& (x := 0))\ ,\ \_\ )$

```
<company>[
  <worker>[
    <surname>m_String:"Durand"
    <name>n_String:"Paul"
    <salary>[ ℓ_Int:6500 ]
  ]
  <worker>[
    <surname>m_String:"Dupond"
    <name>n_String:"Jean"
    <salary>[ℓ_Int:1800]
  ]
  <worker>[
    <surname>m_String:"Martin"
    <name>n_String:"Jules"
    <salary>[ℓ_Int:1200]
  ]
]
```

```
<company>[
  <worker>[
    <surname>m_String:"Durand"
    <name>n_String:"Paul"
  ]
  <worker>[
    <surname>m_String:"Dupond"
    <name>n_String:"Jean"
  ]
  <worker>[
    <surname>m_String:"Martin"
    <name>n_String:"Jules"
  ]
]
```

```
<company>[
  (ℓ_<worker>[Surname Name]:
  <worker>[
    <surname>m_String:"Durand"
    <name>n_String:"Paul"
  ] )
  (ℓ_<worker>[Surname Name]:
  <worker>[
    <surname>m_String:"Dupond"
    <name>n_String:"Jean"
  ] )
]
```

Fig. 4. Labeled XML document and the results of queries Q1 and Q2.

of a labeled function changes as we explained early in this section. Matching clearly is the key rule. Branch selection is performed as in ℂDuce but the labels determined by $/\!\!/$ and indexed by the type of the reductum are prepended to the result. In particular if the first branch is selected, then we just propagate the labels in $v /\!\!/ p_1$, as the second pattern is not even checked. But if the second branch is selected then the result is prepended by both $v /\!\!/ p_2$ *and* $v /\!\!/ \llparenthesis p_1 \rrparenthesis$: while the first set of labels highlights the first of the two kinds of indirect flows present in ℂDuce, namely, those due to pattern matching, the second set of labels fingers the other kind of flows possible in ℂDuce, that is those generated by branch dependency induced by first match policy. Since we selected the second branch e_2 knows that p_1 failed and, therefore, that the matched value v does not belong to (the semantic interpretation of) the type $\llparenthesis p \rrparenthesis$. Therefore, we must also propagate all those labels in v whose content can be (even partially) deduced from the failure of p_1. By definition these are the labels of $v /\!\!/ (\neg \llparenthesis p_1 \rrparenthesis)$ or equivalently (see Lemma 2) of $v /\!\!/ \llparenthesis p_1 \rrparenthesis$.

The reduction semantics we obtain is non deterministic since in case of reduction of a pattern matching or of a labeled function we resort to the type system for determining the type decoration of the reductum's labels (and we know that ℂDuce type system does not enjoy the minimum typing property). As a matter of facts, we have not described a single analysis but a whole family of analyses. This is slightly annoying from a theoretical viewpoint since we have to prove that *all* of them are sound so that, in practice, we can use any of them. However this has no consequence under the practical aspect: first, since we deal with a finite number of labels and types it is always possible to find the best analysis; second, since this problem already resides in the ℂDuce type system a choice was already made by the implementation. So we follow the implementation of ℂDuce and use in Figures 3 and in the operational semantics of ℂDuce$_{\mathcal{L}}$ the types inferred by the ℂDuce's type-checker: we end up with the best analysis (in the family described above) that is sound with the current implementation of ℂDuce.

Let us finally apply the analysis to the queries of Section 1. In order to trace how information of each data flows in the queries, we label the content of each sub-element of <worker> elements by a different label as shown on the left expression of Figure 4. The results of the analysis of Q1 and Q2 respectively are the central and right expressions in Figure 4.[15] Note that Q1 propagates only the labels of name and surname (the

[15] Sequences are encoded by right associative nested pairs ended by 'nil, XML elements <tag *attributes*>s become triples ('tag,(*attributes*,s)), while transform is obtained by iterating matching expressions on the elements, and encapsulating the results into a sequence.

propagation is a consequence of an explicit flow), thus the analysis has correctly indicated that it is safe. The analysis of Q2 instead propagates also the salary label and therefore is rejected as insecure. The propagation of $\ell_{\text{<worker>[Surname Name]}}$ is caused by the "$/\!\!/$" in the first match rule in $\mathbb{C}$Duce$_{\mathcal{L}}$'s operational semantics. In particular when matching the pattern of transform against an element, the third rule for pairs in Figure 3 decomposes the test over each projection, one of which calculates, say in the first loop, $(\ell_{\text{Int}}\text{:6500})/\!\!/\,\text{MoreThanMe}$ which by the second rule for types in Figure 3 is equal to ℓ. Note also how the position of the label denotes different kinds of properties: for example we specified <salary>[ℓ :1200] since we wanted to capture only transformations that depended on the content of the salary element, while if we rather had specified ℓ : <salary>[1200] we would have captured also transformations that test the presence of this element (in the case it were optional).

We want to stress that our analysis can check very complex security entailments. As explained in the introduction, the independence of the result from salary can also be ensured by access control techniques or by stripping salary values from the source document. However, our technique allows one to check that even if a query can access both salaries and names it cannot correlate them. To that end it just suffices to verify that the presence of a m or of a n label in the result implies the absence of the ℓ label, and vice-versa. According to this policy query Q1 would be accepted since ℓ does not occur in the result while query Q2 would be rejected since all the three labels are in the result. A query that plainly returned the list of all salaries (without any name or surname) or some statistic about them, would be considered safe, too. More generally, by using label propagation and some logic (e.g. propositional one) on labels we can define complex security policies whose verification, trivial in our technique, would be very hard (if not impossible at all) by standard access control techniques, as we show in Section 6.

Finally, we can recast the analysis scenarios we outlined in the introduction (Figure 1) to the present setting. A sound static analysis for our system is an analysis that computes labels that will be surely absent from result of the dynamic analysis, while a complete static analysis will determine labels that will be surely present in the same result. Therefore, here completeness is stated with respect to the dynamic analysis rather than with respect to the non-interference property. With that respect the work presented here constitutes the cornerstone of the outlined architecture.

5 Properties

In this section we briefly enumerate the various properties of our approach. For space reasons proofs and less important lemmas are omitted or just sketched. They are all reported in extended version of the article available on-line.

In what follows, when we state that e is a $\mathbb{C}$Duce or $\mathbb{C}$Duce$_{\mathcal{L}}$ expression we implicitly mean that e is a *well-typed* ($\mathbb{C}$Duce or $\mathbb{C}$Duce$_{\mathcal{L}}$) expression.

Lemma 1 (Strip). *Let e be a $\mathbb{C}$Duce$_{\mathcal{L}}$ expression. If $e \rightsquigarrow^* e'$, then $\lfloor e \rfloor \rightsquigarrow^* \lfloor e' \rfloor$.*

This lemma has two important consequences, namely (i) that $\mathbb{C}$Duce$_{\mathcal{L}}$ is a conservative extension of $\mathbb{C}$Duce with respect to the reduction theory, and (ii) that despite being non-deterministic the reduction of $\mathbb{C}$Duce$_{\mathcal{L}}$ preserves the semantics of $\mathbb{C}$Duce programs:

Corollary 1. *Let e be a $\mathbb{C}$Duce$_{\mathcal{L}}$ expression. If $e \rightsquigarrow e'$, and $e \rightsquigarrow e''$, then $\lfloor e' \rfloor = \lfloor e'' \rfloor$.*

The soundness of $\mathbb{C}$Duce$_{\mathcal{L}}$ type system is proved by subject reduction and is instrumental to the soundness of the analysis.

Theorem 1 (Subject reduction). *Let e be a $\mathbb{C}Duce_{\mathcal{L}}$ expression. If $\Gamma \vdash e : t$ and $e \rightsquigarrow^* e'$, then $\Gamma \vdash e' : t$.*

The next lemma is useful to understand the match reduction rules of $\mathbb{C}\mathrm{Duce}_{\mathcal{L}}$.

Lemma 2. *For every value v and type t, $(v /\!\!/ t) = (v /\!\!/ \neg t)$ (modulo the indexes of $\varnothing$)*

In order to prove non-interference we need two lemmas, one that characterizes $/\!\!/$ and a second that is the non-interference counterpart of the standard substitution lemma in typed λ-calculi:

Lemma 3. *Let p be a pattern, v a $\mathbb{C}Duce_{\mathcal{L}}$ value with an occurrence Δ such that $v_\Delta = \ell_t{:}v_1$. If $\ell \notin v /\!\!/ p$, then for all v' such that $\varnothing \vdash v'{:}t$, $(v/p \neq \Omega \Leftrightarrow \mathcal{C}^v_\Delta[v']/p \neq \Omega)$ holds true.*

Lemma 4 (Substitution). *Let p be a pattern, v a $\mathbb{C}Duce_{\mathcal{L}}$ value whose occurrence Δ is such that $v_\Delta = \ell_t : v'$. If ℓ does not occur in the image of v/p and $\ell \notin v /\!\!/ p$, then for all v'' such that $\varnothing \vdash v''{:}t$, we have that $\mathcal{C}^v_\Delta[v'']/p$ is pointwise equal to v/p.*

We can state the non-interference theorem: note that the non-determinism of $\mathbb{C}\mathrm{Duce}_{\mathcal{L}}$'s reduction is accounted for by the quantifying on all results v of the expression e.

Theorem 2 (Non-interference). *Let e be a $\mathbb{C}Duce_{\mathcal{L}}$ expression, with an occurrence Δ such that $e_\Delta = \ell_t : e_1$. For every value v such that $e \rightsquigarrow^* v$, if $\ell \notin \mathcal{L}(v)$, then Δ is non interfering in $\lfloor e \rfloor$ with respect to t, i.e., $\forall v' \in \mathbb{C}Duce$ s.t. $\varnothing \vdash v' : t$, we have $\lfloor \mathcal{C}^e_\Delta[v'] \rfloor \rightsquigarrow^* \lfloor v \rfloor$*

By using Proposition 2 it is easy to see that the conclusion of the theorem implies the Definition 2 of non-interference, justifying in this way the "i.e." we used in the statement of the theorem.

6 A last example

We end our presentation by commenting a more articulated example to illustrate the use of our technique to define and verify complex security policies that cannot be expressed in terms of access control. We suppose to store in XML-documents information about persons that have to pass some examination. The form of the documents is described by the type declarations on the right. As we see every document records a list of names, with personal information, and with an *optional* <grade> element that stores the numerical result of the examination. The absence of such an element denotes that the person has not passed (that is, either not taken or taken and failed) the examination, yet. An XML document that verifies this schema is shown in the left column of Figure 5, while the central column reports the result of importing the same document in $\mathbb{C}$Duce.

```
type ExamBase = <exam_base>[Person*]
type Person =  <person gender="M"|"F">
                        [Name Birth Grade?]
type Name = <name>String
type Birth = <birth>[Year Month Day]
type Year = <year>[Int]
type Month = <month>MName
type MName= "Jan"|"Feb"|"Mar"|"Apr"| · · ·
type Day = <day>[1--31]
type Grade = <grade>[Int]
```

Imagine that examination documents can be accessed by three different categories of users, academic staff, administrative staff, and normal users. We want academic staff to have unconstrained access to the information stored in the examination documents while we may wish to constrain the accesses for administration and normal users. As an example of security requirements we may wish to enforce we have:

1. Only academic users can have information both on names and grades or on names and birthdays simultaneously.

```
<?xml version="1.0"?>
<exam_base>
  <person gender="M">
    <name>Durand</name>
    <birth>
      <year>1970</year>
      <month>Aug</month>
      <day>10</day>
    </birth>
    <grade>110</grade>
  </person>
  <person gender="M">
    <name>Dupond</name>
    <birth>
      <year>1953</year>
      <month>Apr</month>
      <day>22</day>
    </birth>
  </person>
  <person gender="F">
    <name>Dubois</name>
    <birth>
      <year>1965</year>
      <month>Sep</month>
      <day>2</day>
    </birth>
    <grade>120</grade>
  </person>
</exam_base>
```

```
let eb : ExamBase =
<exam_base>[
  <person gender="M">[
    <name>"Durand"
    <birth>[
      <year>[1970]
      <month>"Aug"
      <day>[10]
    ]
    <grade>[110]
  ]
  <person gender="M">[
    <name>"Dupond"
    <birth>[
      <year>[1953]
      <month>"Apr"
      <day>[22]
    ]
  ]
  <person gender="F">[
    <name>"Dubois"
    <birth>[
      <year>[1965]
      <month>"Sep"
      <day>[2]
    ]
    <grade>[120]
  ]
]
```

```
let eb : ExamBase =
<exam_base>[
  <person gender= stat_{"M"|"F"} : "M">[
    <name> name_{String} : "Durand"
    <birth>[
      <year>[ stat_{Int} : 1970]
      <month> private_{MName} : "Aug"
      <day>[ private_{(1--31)} : 10]
    ]
    passed_{Grade} : <grade>[ result_{Int} : 110]
  ]
  <person gender= stat_{"M"|"F"} : "M">[
    <name> name_{String} : "Dupond"
    <birth>[
      <year>[ stat_{Int} : 1953]
      <month> private_{MName} : "Apr"
      <day>[ private_{(1--31)} : 22]
    ]
  ]
  <person gender= stat_{"M"|"F"} : "F">[
    <name> name_{String} : "Dubois"
    <birth>[
      <year>[ stat_{Int} : 1965]
      <month> private_{MName} : "Sep"
      <day>[ private_{(1--31)} : 2]
    ]
    passed_{Grade} : <grade>[ result_{Int} : 120]
  ]
]
```

Fig. 5. A database of examinations: in XML, in ℂDuce, and in ℂDuce$_{\mathcal{L}}$

2. The administrative users can check whether a person passed the examination (that is, they can check for the presence of a <grade> element) but cannot access the result.
3. Every user can ask for statistical results on grades upon criteria limited to year of birth and gender (so that they cannot select sufficiently restrictive sets to infer personal data).

To dynamically verify these constraints we introduce five labels that we use to classify the information stored in documents: *private* (that classifies the month and the day of birth), *stat* (that classifies the year of birth and the gender attribute), *name* (that classifies names), *passed* (that classifies grade elements), and *result* (that classifies the *content* of grade elements). Rather than document-wise, this classification is described directly on types as shown by the definitions on the right. Note that in these definitions labels have no indexes (in documents they will be indexed by the types they are labeling). Before executing a query the system uses this specification to generate a labeled version of the document as shown in the last column of Figure 5. The query is then executed on the labeled document (according to the semantics of ℂDuce$_{\mathcal{L}}$) and the following constraints[16] are checked in the result:

```
type Person = <person gender= stat:("M"|"F")>
                    [ Name Birth ( passed:Grade)? ]
type Name = <name>( name:String)
type Year = <year>[ stat:Int]
type Month = <month>( private:MName)
type Day = <day>[ private:(1--31)]
type Grade = <grade>[ result:Int]
```

If the owner of the query is a normal user, then the result must satisfy:

$$name \Rightarrow \neg\,(private \vee stat \vee result \vee passed) \wedge private \Rightarrow \neg\,(stat \vee result)$$

[16] For the sake of the example we expressed these constraints in a propositional logic where the labels are atoms, but different languages are possible. The definition of such languages is out of the scope of the paper and matter of future research: see Section 7.

if the owner of the query is an administrative user, then the result must satisfy:
$name \Rightarrow \neg\,(private \vee stat \vee result) \wedge private \Rightarrow \neg\,(stat \vee result)$

where a propositional label is satisfied if and only if the label is present in the result. Thus, for instance, the second constraint must be read: if the label *name* occurs in the result then *private*, *stat*, and *result* cannot occur in it, and if *private* occurs in the result then *stat*, and *result* do not. The difference with respect to the constraint for normal users is that the second constraint allows *name* and *passed* to occur simultaneously in the same result. Therefore a query that just tested the presence of a <grade> element without checking its content would satisfy this second constraint.

If the result of a query satisfies the corresponding constraint, then its stripped version is returned to the owner of the query.

7 Perspectives

This paper contains exploratory work toward the definition of information flows security in XML transformations. As such it opens several perspectives both for the practical and the theoretical aspects.

First and foremost the cost of the dynamic analysis must be checked against an implementation (we are currently working on it). We expect this cost to be reasonably low: ℂDuce's pattern matching fully relies on dynamic type checking, therefore if we embed the dynamic generation of typed security labels in ℂDuce's runtime the resulting overhead should be small. Also, to fill the gap with practice we must devise expressive and user-friendly ways to describe the labeling of XML documents in the meta-data (that is, XML Schemas and DTDs) and to express the associated security policies (for instance, in Section 6 we expressed them in propositional logic). These security properties should be defined by sets of constraints (i.e., formulæ of an appropriate logic) that are automatically generated from a specification expressed in an "ad hoc" language (e.g., like the authorizations defined in [6]).

The precision of the dynamic analysis must be enhanced by program rewriting techniques. To exemplify, an obvious "optimization" consists in rewriting all closed (i.e., without capture variables) patterns into an equivalent type constraint pattern, by replacing $\wedge$ for $\&$, $\vee$ for $|$, and $\times$ for $(,)$, so as to transform the pattern $(s\,|\,t,u\&v)$ into the type $(s \vee t) \times (u \wedge v)$. Indeed, the analysis on types is more precise than that on equivalent patterns as the latter is recursively applied to subcomponents forgetting, in doing so, many interdependencies. But other subtler rewritings could improve our analysis: for instance $p_1 = (x\&\mathsf{true})|(x\&\mathsf{false})$ is equivalent to $p_2 = x\&(\mathsf{true} \vee \mathsf{false})$ but $(\ell_{\mathsf{Bool}}{:}e)/\!\!/p_1 = \ell_{\mathsf{Bool}}{::}(e/\!\!/p_1)$ while $(\ell_{\mathsf{Bool}}{:}e)/\!\!/p_2 = \varnothing_{\mathbf{ok}}$. This discrepancy can be identified with the fact that the analyses of pair, intersections, and union patterns are performed independently on the two subpatterns. Thus a possible way to tackle this problem is by transferring some information from one pattern to the other, for example by mimicking the automaton-based technique used by ℂDuce for the just-in-time compilation of patterns.

Finally note that one of the main technical novelties of this work is to endow security labels with constraints. The constraints at issue are quite simple, since they just express the static knowledge of the type of the labeled expression. It is then natural to think of much more expressive constraints. For example we can think of endowing labels with integrity constraints and define non-interference just in terms of consistent databases. In this perspective a program that checked the integrity of a base would be always interference-free even if it accessed private information. Of course checking non-interference in this case would be even more challenging but it could pave the way to (security) proof carrying code for XML.

Acknowledgments: We are very grateful to Nicole Bidoit and Nevin Heintze for their careful reading and invaluable suggestions, and to Dario Colazzo, Alain Frisch, Alban Gabillon, and Massimo Marchiori for their feedback.

References

1. M. Abadi, B. Lampson, and J.-J. Levy. Analysis and caching of dependencies. In *ICFP '96, 1st ACM Conference on Functional Programming*, pages 83–91, 1996.
2. V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General Purpose Language. In *ICFP '03, 8th ACM Conference on Functional Programming*, pages 51–63, 2003.
3. A. Christensen, A. Møller, and M. Schwartzbach. Extending Java for high-level web service construction. *ACM TOPLAS*, 2003. To appear.
4. S. Cluet and J. Siméon. Yatl: a functional and declarative language for XML, 2000. Draft manuscript.
5. S. Conchon. Modular information flow analysis for process calculi. In *FCS 2002, Proceedings of the Foundations of Computer Security Workshop*, Copenhagen, Denmark, 2002.
6. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM TOISS*, 5(2):169–202, 2002.
7. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Design and implementation of an access control processor for XML documents. *Computer Networks*, 33(1–6):59–75, 2000.
8. M. Fernández, J. Siméon, and P. Wadler. An algebra for XML query. In *FST&TCS*, number 1974 in LNCS, pages 11–45, 2000.
9. A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *LICS '02, Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146, 2002.
10. A. Gabillon and E. Bruno. Regulating access to XML documents. In *15th Annual IFIP WG 11.3 Working Conference on Database Security.*, July 15-18 2001.
11. V. Gapayev and B. Pierce. Regular object types. In *Proc. of ECOOP '03*, Lecture Notes in Computer Science. Springer, 2003.
12. J.A. Goguen and J. Meseguer. Security policy and security models. In *Proceedings of Symposium on Secrecy and Privacy*, pages 11–20. IEEE Computer Society, april 1982.
13. H. Hosoya and B. Pierce. XDuce: A typed XML processing language. *ACM TOIT*, 2003. To appear.
14. IBM AlphaWorks. *XML Security Suite*. http://www.alphaworks.ibm.com/tech/xmlsecuritysuite.
15. A. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–142, 1997.
16. F. Pottier and S. Conchon. Information flow inference for free. In *ICFP '00, 5th ACM Conference on Functional Programming*, pages 46–57, September 2000.
17. F. Pottier and V. Simonet. Information flow inference for ML. *ACM SIGPLAN Notices*, 31(1):319–330, January 2002.
18. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
19. D. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97*, number 1214 in Lecture Notes in Computer Science, pages 607–621. Springer, 1997.
20. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
21. C. Wallace and C. Runciman. Haskell and XML: Generic combinators or type based translation? In *ICFP '99, 4th ACM Conference on Functional Programming*, pages 148–159, 1999.