Interprocedural Information Flow Analysis of XML Processors *

Helmut Seidl and Máté Kovács

Technische Universität München, Germany
{seidl,kovacsm}@in.tum.de

Abstract. A crucial issue when providing publicly accessible web services is that sensitive data should only be accessible by authorized users. Accessibility of data within an application or information flow can conveniently be formalized as a 2-hyperproperty of a program. Here, we present a technique to interprocedurally analyze information flow in XML processors. Our approach is based on general techniques for program matching, and relational abstract interpretation of the resulting 2-programs. In case of XML processors, the abstract relational semantics then can be practically analyzed by means of finite tree automata.

1 Introduction

Web services accessed by a variety of users such as a conference management system or the information system of a health insurance company maintain sensible data which should not be accessible to everyone. Therefore, rules are provided to formalize which users are authorized to access which pieces of data. *Information flow analysis* allows to verify that the system complies with the rules given by the access policy by inferring which pieces of observable information may depend on secret data. Such dependencies can conveniently be formalized as *2-hyperproperties* of programs, more precisely, 2-hypersafety properties [6]. 2-Hyperproperties do not refer to single executions of a program but to pairs of executions. Information-flow on the other hand can be formalized by comparing the observations of two executions whose starting states only differ in secret data. No information leaks if unauthorized users cannot observe any difference between the two. This observational indistinguishability has also been called *noninterference* [11].

Consider, e.g., the code fragment in Listing 1 written in the Web Services Business Process Execution Language [1] (BPEL). The fragment is meant to update the health status of patients in a database according to the results of their blood tests. The database is stored in the data structure of variable `pList`. The actual patient is identified by the value in the variable `patientId`, while the update depends on the value of `test`. We consider the blood test and the health

* This work was partially supported by the German Research Foundation (DFG) under the project SpAGAT (grant no. SE 551/14-2) in the priority program "Reliably Secure Software Systems – RS3".

```
<if name="If">
 <condition> <![CDATA[$test < 0.5]]>
 </condition>
  <assign name="EvalGood">
   <copy> <from>"good"</from>
    <to> $pList/patientRecord[id=$patientId]
                /health/text()
    </to> </copy>
  </assign>
 <else>
  <assign name="EvalPoor">
   <copy> <from>"poor"</from>
    <to> $pList/patientRecord[id=$patientId]
                /health/text()
    </to> </copy>
  </assign>
 </else>
</if>
```

Listing 1. A BPEL fragment updating the health status of patients.

status of patients as confidential information, while the rest of the database is public. Accordingly, the data structure is *heterogeneous*, in that it stores confidential as well as public information. Our goal is to prove that the value of `test` only interferes with the health status in the description of the patient, but neither with his name nor with the size of the data structure etc. Such an analysis is challenging, because the control states of the program depend on secret data. In the example, the difference still is not observable, since the two branches perform the same manipulations on public data. However, information flow analyses such as [5, 16, 24] which rely on *tainting* of data elements and program points may not be able to verify the given program. Clarkson and Schneider, on the other hand, have observed that the verification of k-hypersafety properties of a program can be reduced to the verification of ordinary safety properties of the *self-composition* of the program [6]. In this paper we follow the approach of [6] and provide techniques for automatically inferring 2-hypersafety properties by means of *decent* self-compositions of programs. Our observation is that precision can be gained by aligning similar parts of the program and then applying *relational* abstract interpretation.

Related techniques based on the analysis of self-compositions of programs have been presented in [2, 3, 15]. Here, we indicate how this idea can be conveniently extended to languages with procedures. Our exposition is based on the approach of [15] from where we borrow key notions and definitions. The self-composition algorithm there is extended to procedures in such a way that the resulting 2-program can be analyzed by means of standard interprocedural analysis. This means that the two executions performed in parallel are synchronized in a way that they could be implemented on a single runtime stack.

The algorithm uses a *tree distance measure* as a heuristic to align syntactically similar pieces of code fragments with each other. As a result, a call to a procedure *proc* is either aligned with another call to *proc*, or a `skip` instruction. Therefore, in the resulting composition, three procedures are introduced corresponding to *proc* in the original pair of programs, namely, the self-composition [*proc*,*proc*] of *proc*, together with [*proc*,`skip`] and [`skip`,*proc*].

Given a 2-program resulting from the self-composition, we may apply any approach to interprocedural analysis, e.g., the functional approach [9, 14, 21] based on function summaries, or the call string approach advocated in [21] based on abstractions of the call-stack.

In this paper we exemplify the summary-based approach for the case where secrecy is analyzed at the level of program variables. However, in the presence of heterogeneous data structures as in Listing 1 secrecy cannot be successfully analyzed in terms of variables. More generally, web services and business workflows implemented, e.g., in BPEL store data in XML documents and support manipulation of XML documents by means of XPath [4] expressions and XSLT [13] transformations. Accordingly, information flow policies may not only refer to values of variables but may specify the secrecy of individual nodes or subtrees of documents explicitly. In order to discuss these issues, we will not refer to complicated high-level web standards, but consider a simple "assembly" language for tree manipulation where procedures are used, e.g., to evaluate XPath queries. In order to argue relationally about differences of subdocuments, we rely on an abstract domain of *public views* of documents. These are represented by sets of trees while the abstract transformers corresponding to operations on document trees are formalized by means of Horn clauses. The abstract domain of sets of trees seems to be too complicated to allow for concise descriptions of procedure summaries. Therefore, we rely on a simple call string 0 approach where procedure calls and returns are essentially modeled by jumps. Given a 2-program (represented as a control-flow graph), the analysis proceeds in two steps. First, implications are generated to specify the relations of abstract values corresponding to the different nodes in the graph. These implications comply with the format of the subclass of $\mathcal{H}_1$ clauses. The least model of a finite set of such clauses is known to consist of regular sets of trees [18, 25]. Therefore in the second step, the normalization procedure from [18] is applied to compute representations of these sets by means of finite tree automata.

To summarize, this paper has the following contributions:

- An algorithm is introduced for the construction of self-compositions of programs having procedure calls.
- The applicability of our method is demonstrated by proving information flow properties of two examples:
  - In the first example summary functions are used to represent the abstract effect of procedures.
  - In the second example global control flow graphs are applied to prove the information flow security of a tree-manipulating program.

Our exposition uses a minimalistic language in order to simplify the presentation and to emphasize the key features of our techniques.

The rest of this paper is organized as follows. In Section 2 the concepts of the programming language and its semantics are formalized. In Section 3 our technique is introduced to compose pairs of programs, and the relation between programs and their compositions is discussed. Section 4 shows how to apply abstract interpretation on self-compositions of programs. As a first interprocedural analysis, an information flow analysis is presented, which interprocedurally tracks on disequalities of variables using function summarization. The application of global control flow graphs is discussed in Section 5. Finally, in Section 6 we relate our work to others and conclude.

2 Preliminaries

We use a structured programming language extended with procedures in order to present our analyses. We assume that all variables are local. All locals are passed to a called procedure where the result computed by a call is passed to the caller via the dedicated variable `ret`. The syntax of programs f is given by the following grammar:

$$\begin{array}{lll} \text{(declarations)} & f ::= \texttt{procedure}\ \ \mathit{proc}\{p\}\ f \mid p \\ \text{(program)} & p ::= \varepsilon \mid c;p \\ \text{(command)} & c ::= \texttt{skip} \mid x\ \texttt{:=}\ e \mid \texttt{call}\ \ \mathit{proc} \mid \texttt{while}\ b\ \{p\} \mid \\ & \qquad \texttt{if}\ b\ \{p_{\texttt{tt}}\}\ \texttt{else}\ \{p_{\texttt{ff}}\} \end{array} \tag{1}$$

Thus, a program consists of a sequence of procedure declarations, followed by the main program. The main program and the bodies of procedures are sequences of commands. A command c is either an assignment, a procedure call, a conditional selection of branches `if` or a `while` loop. For convenience there is also a `skip` command that leaves the program state intact. Expressions are used on the right-hand-side of assignments to compute new values based on already existing ones. In later sections we will instantiate the expression language for particular examples.

The semantics of programs is defined by means of a transition relation $\mathit{cfg}_1 \to \mathit{cfg}_2$ between configurations. The transitive closure of the transition relation is denoted by $\to^*$. A configuration is a tuple $\langle p, s\rangle$, where p is a sequence of commands to be executed on the state s. We denote the set of states by S. The execution of a program has terminated, if the configuration is of the form $\langle \varepsilon, s\rangle$, i.e., the remaining sequence of commands to be executed is empty. We abbreviate the configuration $\langle \varepsilon, s\rangle$ by the state s.

The semantics of the programming language is shown in Figure 1. The semantics of a call to a procedure *proc*, is defined by means of the functions $\mathsf{enter} : S \to S$ and $\mathsf{combine} : S \times S \to S$. The function enter constructs the initial state for the execution of the procedure, while combine combines the original state at the call site with the return value computed by the procedure. In case the state is a mapping from variables to values, then all variables are passed as

$$\text{A: } \langle x\texttt{:=}e\texttt{;}p, s\rangle \to \langle p, [\![x\texttt{:=}e]\!]s\rangle \qquad \text{SK: } \langle \texttt{skip;}p, s\rangle \to \langle p, s\rangle$$

$$\text{PC: } \frac{\langle p', \mathsf{enter}(s)\rangle \to^* s' \quad \text{where the procedure definition is: } \texttt{procedure } \mathit{proc}\{p'\}}{\langle \texttt{call } \mathit{proc}\texttt{;}p, s\rangle \to \langle p, \mathsf{combine}(s, s')\rangle}$$

$$\text{WT: } \frac{[\![b]\!]s = \texttt{tt}}{\langle \texttt{while } b\ \{p_{\texttt{tt}}\}\texttt{;}p, s\rangle \to \langle p_{\texttt{tt}}\ \texttt{while } b\ \{p_{\texttt{tt}}\}\texttt{;}p, s\rangle}$$

$$\text{WF: } \frac{[\![b]\!]s = \texttt{ff}}{\langle \texttt{while } b\ \{p_{\texttt{tt}}\}\texttt{;}p, s\rangle \to \langle p, s\rangle}$$

$$\text{IT: } \frac{[\![b]\!]s = \texttt{tt}}{\langle \texttt{if } b\ \{p_{\texttt{tt}}\}\ \texttt{else}\ \{p_{\texttt{ff}}\}\texttt{;}p, s\rangle \to \langle p_{\texttt{tt}}\ \texttt{if } b\ \{p_{\texttt{tt}}\}\ \texttt{else}\ \{p_{\texttt{ff}}\}\texttt{;}p, s\rangle}$$

$$\text{IF: } \frac{[\![b]\!]s = \texttt{ff}}{\langle \texttt{if } b\ \{p_{\texttt{tt}}\}\ \texttt{else}\ \{p_{\texttt{ff}}\}\texttt{;}p, s\rangle \to \langle p_{\texttt{ff}}\ \texttt{if } b\ \{p_{\texttt{tt}}\}\ \texttt{else}\ \{p_{\texttt{ff}}\}\texttt{;}p, s\rangle}$$

Fig. 1. The semantics of the programming language.

parameters, and results are communicated back to the caller in the variable `ret`. Therefore, we have:

$$\begin{aligned} \mathsf{enter}(s) &= s \\ \mathsf{combine}(s, s') &= s[\texttt{ret} \mapsto s'(\texttt{ret})] \end{aligned}$$

Above, $s[x \mapsto v]$ denotes the mapping where for all $y \neq x$ it holds that $s[x \mapsto v] = s(y)$ and $s[x \mapsto v](x) = v$.

3 Self-Composition of Programs

The information flow properties we want to prove can be formalized as 2-hypersafety properties [6]. Formally, a 2-hypersafety property is given by two relations on program states, the initial ρ_{in} and final ρ_{fi} relations which specify the pairs of states that should be observationally equivalent. A program satisfies the (end-to-end) 2-hypersafety property, if $(s, t) \in \rho_{in}$ entails that $(s', t') \in \rho_{fi}$ whenever $\langle p, s\rangle \to^* s'$ and $\langle p, t\rangle \to^* t'$. The verification of a hypersafety property of a program on the other hand, can be reduced to the verification of an *ordinary* safety property, but now for a self-composition of the program [3, 6, 23]. Safety properties can be inferred by means of abstract interpretation [8].

Definition 1. *A program pp is a self-composition of the program p if for all pairs of states s and t it holds that whenever* $\langle p, s\rangle \to^* s'$ *and* $\langle p, t\rangle \to^* t'$ *then* $\langle pp, (s, t)\rangle \to^* (s', t')$.

A program satisfying the condition in Definition 1 will be called *2-program*. A 2-program manipulates pairs of states. We construct self-compositions for each

procedure and the main program separately. In the case of branching constructs it is possible that different branches are executed on the two members of a pair of states. Therefore, now we discuss how to construct the composition of two potentially different sequences of commands p and q in general. We construct the composition of two programs using two mutually recursive functions:

- $\mathsf{p2c}(p, q)$ constructs the composition of the two sequences of commands p and q.
- $\mathsf{c2c}(c, d)$ computes the composition of the two commands c and d.

3.1 Composing two Sequences of Commands

The composition of two sequences $p = c_1\,;\ldots;\,c_k$ and $q = d_1\,;\ldots;\,d_l$ is constructed by the function $\mathsf{p2c}(p, q)$. An alignment Ω of the two sequences is a sequence of pairs of commands $(c'_1, d'_1)\,;\ldots;\,(c'_m, d'_m)$, where each c'_i and d'_j are either a `skip` operation or a command occurring in the original sequences. The set of all possible alignments $A(p, q)$ is recursively defined by:

$$\begin{aligned}
A(\varepsilon, \varepsilon) &= \varepsilon \cup \{(\texttt{skip}, \texttt{skip})\,;\Omega \mid \Omega \in A(\varepsilon, \varepsilon)\} \\
A(\varepsilon, d\,;q) &= \{(\texttt{skip}, d)\,;\Omega \mid \Omega \in A(\varepsilon, q)\} \cup \\
&\quad\ \{(\texttt{skip}, \texttt{skip})\,;\Omega \mid \Omega \in A(\varepsilon, d\,;q)\} \\
A(c\,;p, \varepsilon) &= \{(c, \texttt{skip})\,;\Omega \mid \Omega \in A(p, \varepsilon)\} \cup \\
&\quad\ \{(\texttt{skip}, \texttt{skip})\,;\Omega \mid \Omega \in A(c\,;p, \varepsilon)\} \\
A(c\,;p, d\,;q) &= \{(c, d)\,;\Omega \mid \Omega \in A(p, q)\} \cup \\
&\quad\ \{(\texttt{skip}, d)\,;\Omega \mid \Omega \in A(c\,;p, q)\} \cup \\
&\quad\ \{(c, \texttt{skip})\,;\Omega \mid \Omega \in A(p, d\,;q)\} \cup \\
&\quad\ \{(\texttt{skip}, \texttt{skip})\,;\Omega \mid \Omega \in A(c\,;p, d\,;q)\}
\end{aligned} \tag{2}$$

In order to identify a *decent* alignment, we proceed as in [15] by defining the function $\mathsf{p2c}$. This function chooses a best alignment Ω_{opt} according to some *tree distance measure* td for pairs of commands. Accordingly, we set:

$$\Omega_{opt} = \operatorname*{arg\,min}_{\Omega \in A(p,q)} \sum_{i=1}^{|\Omega|} \mathsf{td}(\Omega[i].1, \Omega[i].2)$$

where $\Omega[i]$ stands for the i^{th} pair in the sequence, and $\Omega[i].1$ and $\Omega[i].2$ stand for the first and second members of the pair, respectively. The best alignment Ω_{opt} according to td is the one, where the sum of the distances between the abstract syntax trees of the members of pairs is minimal. In our implementation, we use for td the *Robust Tree Edit Distance* of [19]. Finally the function $\mathsf{c2c}(\Omega_{opt}[i].1, \Omega_{opt}[i].2)$ is called for all pairs of commands in the best alignment in order to construct the corresponding fragment in the composition of the sequences.

3.2 Composing two Commands

In this section we discuss how the composition of two commands is constructed. The composition of two commands is a fragment of a 2-program that operates on pairs of states in order to meet the requirements of Definition 1.

We regard two commands as *composable*, if both of them are either `if` constructs, `while` constructs, `skip` instructions, invocations of the same procedure, or syntactically equal assignments. In our implementation we furthermore require from branching constructs that their conditions are also syntactically equal.

First, we treat the case when the pair of commands are *not* composable. For this we make use of two functions $\mathsf{skip1}(c)$ and $\mathsf{skip2}(c)$. In case c is an assignment, then $\mathsf{skip1}(c) = \texttt{[skip,}c\texttt{]}$ and $\mathsf{skip2}(c) = \texttt{[}c\texttt{,skip]}$. In case c is a call to some procedure *proc*, then we set: $\mathsf{skip1}(c) = \texttt{call [skip,}\mathit{proc}\texttt{]}$ and $\mathsf{skip2}(c) = \texttt{call [}\mathit{proc}\texttt{,skip]}$. In case c is a branching construct, then skip1 replaces the conditional expression b with `[true,`b`]`, skip2 replaces b with `[`b`,true]` and both of them transform the subtrees of the corresponding ASTs recursively. Accordingly, we have:

$$\begin{aligned}
\mathsf{skip1}(\texttt{if } b\ \{p_{\texttt{tt}}\}\ \texttt{else}\ \{p_{\texttt{ff}}\}) &= \texttt{if [true,}b\texttt{]}\ \{\mathsf{skip1}(p_{\texttt{tt}})\}\ \texttt{else}\ \{\mathsf{skip1}(p_{\texttt{ff}})\}\\
\mathsf{skip1}(\texttt{while } b\ \{p\}) &= \texttt{while [true,}b\texttt{]}\ \{\mathsf{skip1}(p)\}\\
\mathsf{skip2}(\texttt{if } b\ \{p_{\texttt{tt}}\}\ \texttt{else}\ \{p_{\texttt{ff}}\}) &= \texttt{if [}b\texttt{,true]}\ \{\mathsf{skip2}(p_{\texttt{tt}})\}\ \texttt{else}\ \{\mathsf{skip2}(p_{\texttt{ff}})\}\\
\mathsf{skip2}(\texttt{while } b\ \{p\}) &= \texttt{while [}b\texttt{,true]}\ \{\mathsf{skip2}(p)\}
\end{aligned}$$

The application of skip1 and skip2 on sequences $p = c_1\texttt{;}c_2\texttt{;},...$ is just syntactic sugar for applying them to each command individually, i.e., $\mathsf{skip1}(c_1\texttt{;}c_2\texttt{;},...) = \mathsf{skip1}(c_1)\texttt{;}\mathsf{skip1}(c_2)\texttt{;}...$

Now we can define the composition of two commands c and d that are not composable by:

$$\mathsf{c2c}(c, d) = \mathsf{skip1}(c)\texttt{;}\mathsf{skip2}(d)$$

In other words, the composition of two commands that are not composable is a sequence of two pairs of commands, where the first manipulates the first member of the pair of states and the second manipulates the second member of the pair of states.

```
1  if [b1,b2] {
2      p2c(ptt, qtt)
3  } else {
4      if [¬b1,b2] {
5          p2c(pff, qtt)
6      } else {
7          if [b1,¬b2] {
8              p2c(ptt, qff)
9          } else {
10             p2c(pff, qff)
11         };
12     };
13 };
```

Listing 2. The composition of two `if` constructs.

Now we consider the case when the two commands are composable. For two equal assignments or procedure calls we have that:

$$\begin{aligned} \mathsf{c2c}(x\texttt{:=}e, x\texttt{:=}e) &= [x\texttt{:=}e, x\texttt{:=}e] \\ \mathsf{c2c}(\texttt{call } proc, \texttt{call } proc) &= \texttt{call } [proc, proc] \end{aligned}$$

The composition of `if` b_1 $\{p_{\mathtt{tt}}\}$ `else` $\{p_{\mathtt{ff}}\}$ and `if` b_2 $\{q_{\mathtt{tt}}\}$ `else` $\{q_{\mathtt{ff}}\}$ is as follows:

As shown by Listing 2, the composition of two `if` constructs consists of three `if` constructs embedded into each other. The idea is that a composition of the bodies of the original branching constructs should be generated for all possible evaluations of the conditional expressions. The composition of `while` b_1 $\{p\}$ and `while` b_2 $\{q\}$ is given by Listing 3. By composing two loops we need to take

```
1 while [b1,b2] { p2c(p,q) };
2 while [¬b1,b2] { skip1(p) };
3 while [b1,¬b2] { skip2(q) };
```

Listing 3. The composition of two `while` loops.

into consideration that it is possible that they are executed a different number of times. Therefore, the first loop at line 1 in Listing 3 handles the case when the original loops execute in synchrony, while the two loops at lines 2 and 3 handle the case when one of the loops has already terminated, but the other has not.

3.3 Handling Procedures

In the composition of sequences of commands we may find commands of the following forms:

$$\begin{aligned} &\texttt{call } [proc, proc] \\ &\texttt{call } [\texttt{skip}, proc] \\ &\texttt{call } [proc, \texttt{skip}] \end{aligned}$$

The first command executes the self-composition of the procedure identified by *proc*, the other two model the case when only one of the pair of executions calls the procedure. Therefore, when constructing the self-composition of the main program, self-compositions of the procedures need to be constructed as well where each procedure of the original program gives rise to three procedures in the resulting 2-program. For each procedure declaration `procedure` $proc\{p\}$ we generate:

$$\begin{aligned} &\texttt{procedure } [proc, proc]\{\mathsf{p2c}(p,p)\} \\ &\texttt{procedure } [\texttt{skip}, proc]\{\mathsf{skip1}(p)\} \\ &\texttt{procedure } [proc, \texttt{skip}]\{\mathsf{skip2}(p)\} \end{aligned}$$

The body of the procedure `[`*proc*`,`*proc*`]` is the self-composition of the body of the original procedure. The other two take care of the case when only one member of a pair of commands is a procedure call and the other is `skip`.

3.4 Semantics of Self-Compositions of Programs

For pairs of Boolean expressions, we define $[\![\texttt{[}b_1\texttt{,}b_2\texttt{]}]\!](s,t) = [\![b_1]\!]s \wedge [\![b_2]\!]t$. In case of assignments c or `skip` we set $[\![\texttt{[}c\texttt{,}d\texttt{]}]\!](s,t) = ([\![c]\!]s, [\![d]\!]t)$. In case of procedure calls the semantics is defined analogously to the original semantics of procedures. Given that a procedure declaration in the composition is `procedure` r $\{pp\}$ we define $[\![\texttt{call } r]\!]^\sharp(s,t) = \mathsf{combine}_2((s,t),(s',t'))$ where $\langle pp, \mathsf{enter}_2(s,t)\rangle \rightarrow^* (s',t')$. If both states are mappings from variables to values, they are given by:

$$\begin{aligned}&\mathsf{enter}_2(s,t) = (s,t)\\ &\mathsf{combine}_2((s,t),(s',t')) = (s[\texttt{ret} \mapsto s'(\texttt{ret})], t[\texttt{ret} \mapsto t'(\texttt{ret})])\end{aligned}$$

In general, the functions enter_2, $\mathsf{combine}_2$ should meet the following requirements:

$$\begin{aligned}\mathsf{enter}_2(s,t) &= (\mathsf{enter}(s), \mathsf{enter}(t))\\ \mathsf{combine}_2((s,t),(s',t')) &= (\mathsf{combine}(s,s'), \mathsf{combine}(t,t'))\end{aligned}$$

Finally, we define the 2-program $\mathsf{p2c}(f,f)$ for a program f as the collection of procedure declarations for `[`$proc$`,`$proc$`]`, `[skip,`$proc$`]`, `[`$proc$`,skip]` ($proc$ declared in f), together with $\mathsf{p2c}(p,p)$ for the main program p of f. Then the following relation between the semantics of a sequence of commands and the self-composition which we have constructed can be proven by induction on the abstract syntax:

Theorem 1. *Any sequence p of commands of a program f and its self-composition $\mathsf{p2c}(p,p)$ satisfy the condition of Definition 1.* □

By Theorem 1, the information flow analysis of an ordinary program f with procedures can be reduced to inferring interprocedural invariants for the 2-program $\mathsf{p2c}(f,f)$. In the next section we review two examples of such interprocedural analyses.

4 Relational Interprocedural Analysis of Self-Compositions of Programs

In order to analyze the self-composition of a program it is convenient to represent the main program as well as the declared procedures of the self-composition by means of control flow graphs (CFG). These CFGs differ from CFGs for ordinary programs only in that edges are not labeled with single assignments, guards or procedures, but with *pairs* of such actions. Thus, one such CFG is a tuple $G = (N, E, n_{in}, n_{fi})$, where N is a finite set of nodes, E is a set of directed and labeled edges, and n_{in} and n_{fi} are the initial and final nodes, respectively. The nodes of the CFG stand for sequences of (pairs of) commands pp that need to be executed on states in a configuration $\langle pp, (s,t)\rangle$, while the edges of the CFG represent the transition relation $\rightarrow$. Whenever $\langle c; pp, (s,t)\rangle \rightarrow \langle pp, (s',t')\rangle$ then the corresponding edge of the CFG is labeled by c if the command is an assignment or a call. Guards bb and $\neg bb$ are used as labels for edges entering the then and

else branches of `if` commands with condition bb, and accordingly, for entering the bodies of `while` loops with condition bb or exiting those loops, respectively. In the concrete semantics, each label l of an edge corresponds to a state transformer $[\![l]\!] : S \times S \to S \times S$. These state transformers are partial functions, i.e., the guards that hold on a pair of states propagate the pair unmodified, while the result is undefined on those states on which the guard does not hold.

In order to perform abstract interpretation for self-compositions of programs, we choose a complete lattice $(\mathbb{D}, \sqsubseteq)$ whose elements represent the potential invariants at program points. We use a function γ to map an abstract value $D \in \mathbb{D}$ to the set pairs of states for which the invariant D holds. As usual, the function γ is called *concretization*.

Example 1. Assume that we only want to track the set of program variables where the two states possibly differ. For that, we choose $\mathbb{D}$ as the complete lattice $\mathcal{P}(\mathcal{X})_\bot = \{\bot\} \cup \mathcal{P}(\mathcal{X})$ consisting of a least element $\bot$ (denoting unreachability) together with all subsets of the set of program variables $\mathcal{X}$. The ordering relation is given by $\bot \sqsubseteq D$ for all $D \in \mathbb{D}$, and for $D \neq \bot \neq D'$, $D \sqsubseteq D'$ iff $D \subseteq D'$.

The concretization $\gamma(V)$ of an abstract value V is then defined by:

$$\begin{aligned} \gamma(V) &= \{(s,t) \mid \forall x \in \mathcal{X} : x \notin V \Rightarrow s(x) = t(x)\} \\ \gamma(\bot) &= \emptyset \end{aligned} \tag{3}$$

Besides a complete lattice of potential invariants, transfer functions $[\![l]\!]^\sharp$ are needed for the labels l at control flow edges, which describe how the potential invariants are affected. For a transfer function $[\![l]\!]^\sharp$, we require that for all abstract values D and pairs of states (s,t) with $(s,t) \in \gamma(D)$ the following holds:

$$[\![l]\!](s,t) \in \gamma([\![l]\!]^\sharp D)$$

Example 2. In the case when only potential disequalities are tracked, all transfer functions $[\![l]\!]^\sharp$ preserve the abstract value $\bot$. For $V \neq \bot$, we could have:

$$\begin{aligned} [\![x\texttt{:=}e, x\texttt{:=}e]\!]^\sharp(V) &= (\mathsf{vars}(e) \cap V \neq \emptyset)\,?\,(V \cup \{x\}) : (V \backslash \{x\}) \\ [\![x\texttt{:=}e, \texttt{skip}]\!]^\sharp(V) &= V \cup \{x\} \\ [\![\texttt{skip}, x\texttt{:=}e]\!]^\sharp(V) &= V \cup \{x\} \\ [\![\neg b, b]\!]^\sharp(V) &= (\mathsf{vars}(b) \cap V \neq \emptyset)\,?\,V : \bot \\ [\![b, \neg b]\!]^\sharp(V) &= (\mathsf{vars}(b) \cap V \neq \emptyset)\,?\,V : \bot \\ [\![bb]\!]^\sharp(V) &= V \qquad \text{for any other guard } bb \end{aligned}$$

Here, we use the "? :" operator as in C for representing a conditional choice, i.e.,

$$b\,?\,x : y = \begin{cases} x & \text{if } b \text{ holds} \\ y & \text{otherwise} \end{cases}$$

Note that all listed transformers are indeed *monotonic*. The intuition behind these definitions are as follows. Whenever both assignments are syntactically

equal, then the same value will be assigned to the left-hand side variable x, if all variables occurring on the right-hand side agree in their respective values. In this case, x must be removed from the argument set V. Otherwise, it must be added. On the other hand when guards are concerned, we can be sure that the condition is not met only when the guard is of the form $\texttt{[}b\texttt{,}\neg b\texttt{]}$ or $\texttt{[}\neg b\texttt{,}b\texttt{]}$, and all variables occurring in b agree in their respective values. In this case, the guard definitely cannot be taken and therefore $\bot$ can be returned. In all other cases, no information can be extracted.

Besides transformers for assignments and guards, we can also provide abstract enter and $\mathsf{combine}$ functions. For the analysis of Example 1, both functions preserve $\bot$ (in each argument); for non-$\bot$ arguments, their results are given by:

$$\begin{aligned}\mathsf{enter}_2{}^\sharp(V) &= V\\ \mathsf{combine}_2{}^\sharp(V_1,V_2) &= (V_1 \setminus \{\texttt{ret}\}) \cup (V_2 \cap \{\texttt{ret}\})\end{aligned}$$

Since the function enter_2 simply forwards the state to the called procedure, its abstract variant is the identity function as well. Because the values of variable `ret` in the resulting pair of concrete states only depend on the result of the procedure call, it is first removed from the initial abstract value V_1. In case the results of a pair of procedure calls may differ in the variable `ret`, then this variable may contain different values after evaluating $\mathsf{combine}_2$. Therefore, its abstract variant adds `ret` to the set of variables possibly affected by the secret.

Although we are thus given abstract versions of enter_2 and $\mathsf{combine}_2$, the abstract transformers for call edges are still not yet known: they additionally depend on the abstract semantics of the called procedures. In order to compute these, we introduce for each program point n of a CFG the unknown $[\![n]\!]^\sharp$, which denotes the transformer that transforms the abstract value at the entry point of the DFG into the abstract value at program point n. These abstract transformers can jointly be approximated by the least solution of the following constraint system:

$$\begin{array}{lll}
[\![n_{in}]\!]^\sharp \sqsupseteq \mathsf{Id} & & \text{For the initial node } n_{in} \text{ of some CFG.}\\
[\![n]\!]^\sharp \sqsupseteq H^\sharp([\![n^r_{fi}]\!]^\sharp) \circ [\![m]\!]^\sharp & & \text{For an edge: } (m, \texttt{call } r, n) \quad \text{where } n^r_{fi} \text{ is the}\\
& & \text{final node of the CFG of procedure } r.\\
[\![n]\!]^\sharp \sqsupseteq [\![l]\!]^\sharp \circ [\![m]\!]^\sharp & & \text{For an edge: } (m,l,n) \quad \text{where } l \text{ is a guard or}\\
& & \text{an assignment.}
\end{array} \tag{4}$$

Here, Id denotes the identity function, $\circ$ denotes function composition and $H^\sharp$ is an operator which takes the effect of a procedure body and transforms it into the effect of a call. Thus, $H^\sharp$ is defined by:

$$H^\sharp\ g\ D = \mathsf{combine}_2{}^\sharp(D, g(\mathsf{enter}_2{}^\sharp(D)))$$

This constraint system does no longer speak about the complete lattice $\mathbb{D}$ itself but about the set of monotonic functions in $\mathbb{D} \to \mathbb{D}$. This set, however, can again be considered as a complete lattice w.r.t. the argumentwise ordering on

functions. Likewise, if $\mathbb{D}$ is finite (as in our example) then also the set $\mathbb{D} \to \mathbb{D}$ is finite. Therefore, the least fixpoint solution of the constraint system can be computed by ordinary Kleene iteration.

Now assume that we are given a set V_0 of variables which at program start may contain secrets. Furthermore, assume that h is the abstract transformer computed by the analysis for the exit point of the main program. If $h(V_0) = \bot$, then we can be sure that program exit cannot be simultaneously reached by two executions starting with initial states $(s_0, t_0) \in \gamma(V_0)$. If $h(V_0) \neq \bot$, then all variables that may potentially leak information about the initial secrets are contained in the set $h(V_0)$.

```
1 procedure fib {
2    x := xin;
3    if x = 0 { ret := 0; }
4    else {
5       if x = 1 { ret := 1; }
6       else {
7          one := 1;
8          xin := x-one;
9          call fib;
10         r1 := ret;
11         xin := xin-one;
12         call fib;
13         r2 := ret;
14         ret := r1+r2;
15      };
16   };
17 }
18
19 xin := secin;
20 call fib;
21 secfib := ret;
22 xin := pubin
23 call fib;
24 pubfib := ret;
```

Listing 4. A program computing Fibonacci numbers.

Let us, for example, consider the recursive program from Listing 4 which computes two members of the Fibonacci series from the numbers in variables `secin` and `pubin` of the initial state, respectively. The corresponding two return values then are stored in the variables `secfib` and `pubfib` of the final state, respectively.

```
1 procedure [skip,fib] {
2    [skip,x := xin];
3    if [true,x = 0} { [skip,ret := 0]; }
4    else {
5       if [true,x = 1] { [skip,ret:= 1]; }
6       else {
7          [skip,one := 1];
8          [skip,xin := x-one];
```

```
          call [skip,fib];
          [skip,r1 := ret];
          [skip,xin := x-one];
          call [skip,fib];
          [skip,r2 := ret];
          [skip,ret := r1+r2];
       };
    };
}

procedure [fib,skip] {
    [x := xin,skip];
    if [x = 0,true] { [ret := 0,skip]; }
    else {
       if [x = 1,true] { [ret := 1,skip]; }
       else {
          [one := 1,skip];
          [xin := x-one,skip];
          call [fib,skip];
          [r1 := ret,skip];
          [xin := x-one,skip];
          call [fib,skip];
          [r2 := ret,skip];
          [ret := r1+r2,skip];
       };
    };
}

procedure [fib,fib] {
    [x:=xin,x:=xin];
    if [x = 0,x = 0] {  [ret := 0,ret := 0];
    } else {
       if [¬ x=0, x=0] {
          [skip,ret:=0];
          if [x = 1,true] { [ret := 1,skip]; }
          else {
             [one := 1,skip];
             [xin := x-one,skip];
             call [fib,skip];
             [r1 := ret,skip];
             [xin := xin-one,skip];
             call [fib,skip];
             [r2 := ret,skip];
             [ret := r1+r2,skip];
           };
       } else {
          if [x=0, ¬ x=0] {
             [ret := 0,skip]
             if [true,x = 1] { [skip,ret:= 1]; }
             else {
```

```
59                [skip,one := 1];
60                [skip,xin := x-one];
61                call [skip,fib];
62                [skip,r1 := ret];
63                [skip,xin := xin-one];
64                call [skip,fib];
65                [skip,r2 := ret];
66                [skip,ret := r1+r2];
67             };
68          } else {
69             if [x = 1,x = 1] { [ret := 1,ret := 1]; }
70             else {
71                if [¬ x=1,x = 1] {
72                   [skip,ret := 1];
73                   [one := 1,skip];
74                   [xin := x-one,skip];
75                   call [fib,skip];
76                   [r1 := ret,skip];
77                   [xin := xin-one,skip];
78                   call [fib,skip];
79                   [r2 := ret,skip];
80                   [ret := r1+r2,skip];
81                } else {
82                   if [x = 1,¬ x=1] {
83                      [ret:= 1,skip];
84                      [skip,one := 1];
85                      [skip,xin := x-one];
86                      call [skip,fib];
87                      [skip,r1 := ret];
88                      [skip,xin := xin-one];
89                      call [skip,fib];
90                      [skip,r2 := ret];
91                      [skip,ret := r1+r2];
92                   } else {
93                      [one := 1,one := 1];
94                      [xin := x-one,xin := x-one];
95                      call [fib,fib];
96                      [r1 := ret,r1 := ret];
97                      [xin := xin-one,xin := xin-one];
98                      call [fib,fib];
99                      [r2 := ret,r2 := ret];
100                     [ret := r1+r2,ret := r1+r2];
101                  };
102               };
103            };
104         };
105      };
106   };
107}
108
```

```
109 [xin := secin,xin := secin];
110 call [fib,fib];
111 [secfib := ret,secfib := ret];
112 [xin := pubin,xin := pubin];
113 call [fib,fib];
114 [pubfib := ret,pubfib := ret];
```

Listing 5. The self-composition of the program in Listing 4.

The self-composition of the program in Listing 4 is shown in Listing 5.

According to the abstract semantics, we can compute the abstract transformers for the end points of the procedures `[fib,fib]`, `[skip,fib]`, and `[fib,skip]`. Clearly, all of them map the bottom element $\bot$ to $\bot$. Furthermore, whenever `xin` is not contained in the argument set V, then `ret` will not be member of resulting abstract value. On the other hand if `xin` is member of the argument V, then `ret` will also be member of the resulting abstract value. As a consequence, the secret value in `secin` does only influence the variable `secfib` at program exit, but does not influence the variable `pubfib`.

5 Analysis Using Global Control Flow Graphs

Summary based interprocedural analysis may be difficult if the complete lattice $\mathbb{D}$ for expressing relational invariants is infinite. An example for this is the analysis of XML processing programs from [15]. Still, this analysis can be extended to programs with procedures by directly abstracting the occurring call-stacks. Here, we only consider the simplest variant of this approach where each call-stack is abstracted by a call site, i.e., the topmost element of the stack. For each node $n \in N$, we then introduce an unknown $D[n]$ representing the abstract value from $\mathbb{D}$ describing an invariant satisfied by all pairs (s,t) of concrete states reaching node n. In order to determine these values, we again put up a constraint system consisting of the following constraints:

$$\begin{array}{ll} D[n_{in}] \sqsupseteq D_0 & \\ D[n] \quad \sqsupseteq [\![l]\!]^\sharp(D[m]) & \text{For each edge } (m,l,n) \text{ where } l \neq \texttt{call } r. \end{array}$$

Here, D_0 is the abstract value for pairs of initial states, and l is not a call edge. Moreover for every call edge $(m, \texttt{call } r, n)$, we add the constraints:

$$\begin{array}{ll} D[n^r_{in}] \sqsupseteq \mathsf{enter}_2{}^\sharp(D[m]) \\ D[n] \quad \sqsupseteq \mathsf{combine}_2{}^\sharp(D[m], D[n^r_{fi}]) \end{array}$$

Thus at the price of potential loss in precision, it suffices to compute with values from $\mathbb{D}$ alone — without resorting to abstract lattices of functions.

Let us apply this approach to an interprocedural information flow analysis of programs manipulating tree-structured data. For the sake of simplicity our programming language is designed to manipulate binary trees. There is no loss of generality involved, since XML documents can be encoded as binary trees, e.g.,

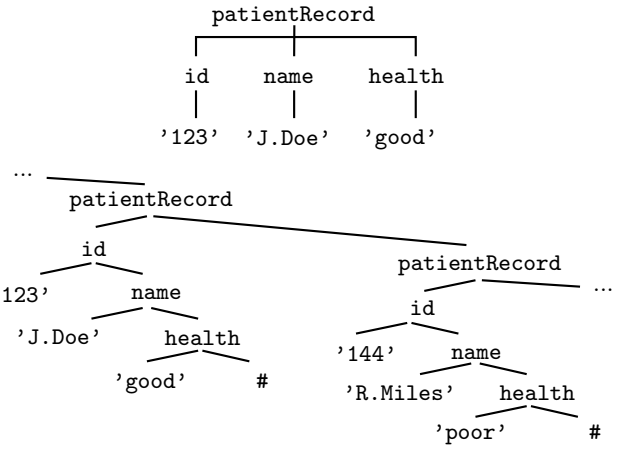

Fig. 2. A record describing a patient in unranked form (top), and its binary representation as a member of a list (bottom).

by means of the *first-child-next-sibling* (FCNS) [7] encoding. Figure 2 shows an unranked tree describing a patient in a database, and its binary representation as member of a list. The FCNS encoding maps the first child of a node to its first child, and its sibling to the right to its second child. The symbol `#` stands for empty unranked forests. It is a requirement that the set of labels of binary and nullary nodes are disjunct. In order to achieve this we put labels of nullary nodes other than `#` between `'` signs and call them *basic values*. We denote the set of labels for binary nodes with Σ_2 and the set of labels for nullary nodes with Σ_0.

$$\begin{array}{ll} \text{(tree expression)} & e ::= \texttt{\#} \mid x \mid x\texttt{/1} \mid x\texttt{/2} \mid \sigma_2\texttt{(}x, y\texttt{)} \mid \lambda_t\texttt{(}x_1, x_2, ...\texttt{)} \\ \text{(Boolean expression)} & b ::= \texttt{top(}x\texttt{)=}\sigma \mid \lambda_b\texttt{(}x_1, x_2, ...\texttt{)} \end{array} \tag{5}$$

The instantiation of the expression language of the programming language shown in (1) can be found in (5). Now there are tree expressions on the right-hand-sides of assignments. A tree expression can be the binary representation of an empty forest `#`, the value of a variable x, the first or second child of the tree in a variable denoted as x`/1` or x`/2` respectively, and a new tree σ_2`(`x`,`y`)` composed of two existing trees and a binary alphabet element. λ_t stands for an interpreted tree function which is provided by the execution environment returning a basic value. A Boolean expression can either check whether the root of a tree is labeled with a specific alphabet element, or it can be an interpreted Boolean expression λ_b provided by the execution environment. An example for an interpreted Boolean expression is `test < '0.5'`. The value of the expression is true if the tree-structured value in the variable `test` consists of a single leaf labeled with a string that can be interpreted as a floating-point number, and its value is less then 0.5.

The execution state s of the programming language typically is a mapping $s : \mathcal{X} \to \mathfrak{B}_{\Sigma_2,\Sigma_0} \cup \{\lightning\}$ from the set of variables $\mathcal{X}$ to the set of binary trees $\mathfrak{B}_{\Sigma_2,\Sigma_0}$ over the binary alphabet Σ_2 and nullary alphabet Σ_0. It can, however, also equal to $\lightning$ — flagging that a runtime error occurred. This happens when the child of a nullary node is queried by an expression of the form $x/\texttt{1}$ or $x/\texttt{2}$.

$$\begin{gathered}
\text{for all } e \text{ it holds that } [\![x\texttt{:=}e]\!]\lightning = \lightning \\
[\![x\texttt{:=}y]\!]s = s[x \mapsto s(y)] \qquad [\![x\texttt{:=}\#]\!]s = s[x \mapsto \#] \\
[\![x\texttt{:=}\sigma\texttt{(}x_1\texttt{,}x_2\texttt{)}]\!]s = s[x \mapsto \sigma(s(x_1), s(x_2))] \\
[\![x\texttt{:=}y\texttt{/1}]\!]s = \begin{cases} s[x \mapsto \tau_1] & \text{if } s(y) = \sigma(\tau_1, \tau_2) \text{ for some label } \sigma \\ & \text{and trees } \tau_1 \text{ and } \tau_2 \\ \lightning & \text{otherwise} \end{cases} \\
[\![x\texttt{:=}y\texttt{/2}]\!]s = \begin{cases} s[x \mapsto t_2] & \text{if } s(y) = \sigma(\tau_1, \tau_2) \text{ for some label } \sigma \\ & \text{and trees } \tau_1 \text{ and } \tau_2 \\ \lightning & \text{otherwise} \end{cases} \\
[\![x\texttt{:=}\lambda_t\texttt{(}x_1\texttt{,}x_2\texttt{,...)}]\!] = s[x \mapsto [\![\lambda_t]\!](s(x_1), s(x_2), ...)]
\end{gathered} \tag{6}$$

The formal semantics of assignments is shown in (6). The error state is propagated by assignments without modification.

Now we define the semantics of Boolean expressions. $[\![\texttt{top(}x\texttt{)=}\sigma]\!]s = s$ if $s(x) = \sigma(\tau_1, \tau_2)$ for some trees τ_1 and τ_2, or if $\sigma = \#$ and $s(x) = \#$. The basic values of leaves can be examined using interpreted Boolean expressions. The fact whether $[\![\lambda_b\texttt{(}x_1\texttt{,}x_2\texttt{,...)}]\!]s = s$ holds, i.e., if an interpreted Boolean expression holds on a state depends on the predefined semantics in the runtime environment.

Boolean expressions propagate the error state only if they are negated, i.e., they are of the form $\neg b$. The results of positive Boolean expressions on the error state is undefined. This behavior is necessary in order to ensure that loops terminate in case an error occurs.

The only requirement for interpreted functions λ_t and λ_b is that they are deterministic, and do not result in the error state.

We define now the semantics of procedure calls. By entering a procedure the values of variables are unchanged. However, if the state is erroneous, then the procedure is not executed. Therefore we have:

$$\mathsf{enter}(s) = \begin{cases} s & \text{if } s \neq \lightning \\ & \text{otherwise undefined} \end{cases}$$

By returning from a procedure execution there are two possibilities. If there was no error during the procedure, then the value of variable `ret` is updated with that at the end of the procedure execution, the values of other variables are propagated from the call site. On the other hand, if there was an error during the execution of the procedure, then the error state needs to be propagated. Therefore we have:

$$\mathsf{combine}(s, s') = \begin{cases} s[\texttt{ret} \mapsto s'(\texttt{ret})] & \text{if } s \neq \lightning \text{ and } s' \neq \lightning \\ \lightning & \text{otherwise} \end{cases}$$

As an example consider the program from the introduction, now compiled to our assembly language. The program is meant to manipulate the database of an insurance company. The database consists of a list of records as it is shown on the bottom of Figure 2. The program updates the health status of patients depending on the results of blood tests as shown by Listing 6.

```
1 procedure query {
2    empty := #;
3    found := false(empty,empty);
4    while top(found) = false {
5       ret := pList/1;
6       idVal := ret/1;
7       if idVal = patientId {
8         found := true(empty,empty);
9       } else {
10         pList := pList/2;
11      };
12   };
13 }
14
15 procedure update {
16   empty :=#;
17   ret := #;
18   newEntIdVal := newEntry/1;
19   while top(pList) = patientRecord {
20      entry:= pList/1;
21      entIdVal := entry/1;
22      if entIdVal = newEntIdVal {
23         ret := patientRecord(newEntry,ret);
24      } else {
25         ret := patientRecord(entry,ret);
26      };
27      pList := pList/2;
28   };
29 }
30
31 procedure createNewEntry{
32   idVal  :=  oldEntry/1;
33   name  :=  oldEntry/2;
34   nameVal  :=  name/1;
35
36   newHealth  :=  health(newHealthVal,empty);
37   newName  :=  name(nameVal,newHealth);
38   ret  :=  id(idVal,newName);
39 }
40
41 empty  :=  #;
42 if test < 0.5 {
43   call query;
44   entry  := ret;
```

```
45    newHealthVal :=  'good';
46    call createNewEntry;
47    newEntry := ret;
48    call update;
49    pList  := ret;
50 } else {
51    call query;
52    entry  := ret;
53    newHealthVal  :=  'poor';
54    call createNewEntry;
55    newEntry := ret;
56    call update;
57    pList := ret;
58 };
```

Listing 6. A routine for updating the database of a hospital.

The main program starts at line 41. At line 42 a branching decision is made based on the result of the blood test. In both branches, first the database entry with the appropriate identifier is queried using the procedure `query`. The procedure iterates over the list of entries in the database and returns the one with the appropriate identifier. Then, a new entry is constructed using the procedure `createNewEntry`, where the health status is updated according to the result of the test. And finally, the old entry is exchanged with the new one in the database using the procedure `update`.

In our example we consider the result of the blood test and the health status of patients as confidential. Therefore, the goal of our analysis is to prove that these values do not interfere with the names and identifiers of patients, the shape and size of the database etc.

We use *public views* of documents in order to identify subdocuments possibly depending on confidential data. Figure 3 shows a pair of documents and a

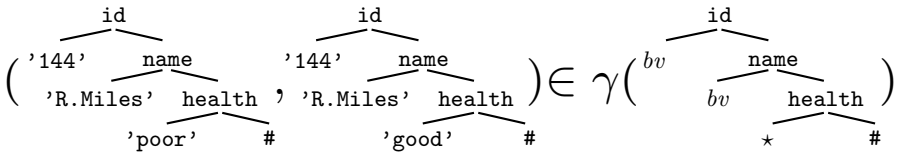

Fig. 3. A pair of documents and a public view of them.

corresponding public view. Thus, we consider a public view as an abstraction of pairs of trees. A pair of trees (τ_1, τ_2) is a member of the concretization $\gamma(\tau)$ of a public view τ, if τ can be constructed from the pair of concrete trees by replacing potentially different subtrees with a leaf labeled $\star$, and by replacing leaves labeled with equal basic values with leaves labeled bv. Accordingly, a public view is a tree over the binary alphabet Σ_2 and the nullary alphabet $\{\#, bv, \star\}$.

An abstract description of a relation of states is a tuple (T, E), where $T : \mathcal{X} \to P(\mathfrak{B}_{\Sigma_2,\{\#,bv,\star\}})$ is a mapping from variables to sets of public views, and E is an element of the 3-point lattice $\{\checkmark \sqsubset \lightning \sqsubset \top\}$. E is necessary in order to represent potential runtime errors. A pair of states (s, t) is a member of the relation given by $(T, \checkmark)$ (denoted[1] $(s, t) \in \gamma(T, \checkmark)$) whenever for each variable $x \in \mathcal{X}$ there is a $\tau \in T(x)$ so that $(s(x), t(x)) \in \gamma(\tau)$. In case $E = \lightning$, then $(\lightning, \lightning)$ is also member of the relation. This represents the situation when a runtime error could have potentially occurred independently of the secret. On the other hand, $E = \top$ represents that a runtime error might have occurred *depending* on the secret. Therefore, in this case pairs of states are also members of the relation where only one member is the error state $\lightning$.

Given an initial abstract value $(T_0, \checkmark)$ describing the set of public views of pairs of potential initial states, we are interested in computing the sets of potential public views for every reachable node of the CFG of the self-composition of the program. In our analysis, sets of public views for variables x occurring at a node n are described by means of unary predicates $\mathtt{var}_{x,n}$, which are defined by means of Horn clauses. Formally, $\tau \in T(x)$ at node n if $\mathtt{var}_{x,n}(\tau)$ holds. The value of E at node n is represented by means of the unary predicate $\mathtt{error}_n$.

Horn clauses are used for the specification of the analysis in two ways. First, they are used to specify the set of potential initial public views for each variable at the initial node n_{in}. Secondly, they are used to formalize how the views at different program points are related to each other. These clauses are obtained from the CFG of the 2-program.

For our analysis, we assume that information flow policies are defined by *regular sets* of public views, and thus can be described by finite tree automata. We will not define finite tree automata here, but note that their languages can be defined by means of clauses of the form:

$$\mathtt{p}(\sigma_0)\,. \quad \text{or} \quad \mathtt{p}(\sigma_2(X_1,X_2)) \Leftarrow \mathtt{p}_1(X_1),\mathtt{p}_2(X_2)\,.$$

Here, $\mathtt{p}$, $\mathtt{p}_1$ and $\mathtt{p}_2$ are unary predicates corresponding to the states of the automaton, and σ_0 and σ_2 are nullary and binary constructors respectively.

In our running example, the health status of patients in the list of records seen in Figure 2 is confidential. Thus, the information flow policy, the set of public views of all possible databases is given by the predicate `qPlist` defined by the implications in (7):

$$\begin{aligned}
\mathtt{qPlist(patientRecord(L,R))} &\Leftarrow \mathtt{qId(L),qPlist(R)}\,.\\
\mathtt{qPlist(\#)}\,. & \\
\mathtt{qId(id(L,R))} &\Leftarrow \mathtt{qBV(L),qName(R)}\,.\\
\mathtt{qName(name(L,R))} &\Leftarrow \mathtt{qBV(L),qHealth(R)}\,.\\
\mathtt{qHealth(health(L,R))} &\Leftarrow \mathtt{qStar(L),qEmpty(R)}\,.\\
\mathtt{qBV}(bv)\mathtt{.\ qStar}(\star)\mathtt{.\ qEmpty(\#)}\,. &
\end{aligned} \tag{7}$$

[1] Here we overload the notation γ and use it for the concretization of public views of document trees, and for the concretization of abstract descriptions of pairs of states.

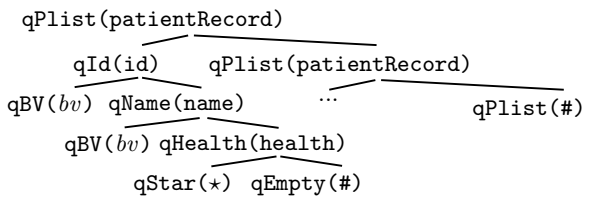

Fig. 4. A run of the automaton specified in (7) accepting the public view of a list of records like that on the bottom of Figure 2 with state `qPlist`.

As Figure 4 illustrates, a list accepted by the predicate `qPlist` is either a tree with root labeled `patientRecord` having a first child accepted by `qId` and a second child accepted by `qPlist`, or it is a nullary node with label `#`. The other predicates can be understood similarly. In particular, the predicate `qHealth` accepts a tree only, if the first child of its root is a leaf labeled $\star$ specifying that the corresponding value is confidential. Supposing that the variable `pList` contains the database in the initial state of our program, the corresponding information for the initial node n_{in} is defined by: $\mathtt{var}_{\mathtt{pList},n_{in}}\mathtt{(X)} \Leftarrow \mathtt{qPlist(X)}.$

In the following, we describe how the abstract state transformers $[\![l]\!]^\sharp$ of edges (m, l, n) are formalized by means of Horn clauses. In order to do so we need that the set of binary elements Σ_2 potentially occurring in the program is finite and a priori known. This information can be extracted, e.g., from the interface descriptions of web services. Due to the construction of self-compositions of programs edges either refer to assignments or to Boolean expressions, but never to both.

5.1 Horn Clauses for Procedure Calls

Consider a call edge $(m, \mathtt{call}\ \ r, n)$ in one of the CFGs of the program. Here we denote the initial and final nodes of the CFG corresponding to procedure r with n^r_{in} and n^r_{fi}, respectively.

The function enter_2 propagates values of variables without modification. Therefore, we generate the following implication for each variable x:

$$\mathtt{var}_{x,n^r_{in}}\mathtt{(X)} \Leftarrow \mathtt{var}_{x,m}\mathtt{(X)}.$$

For the propagation of the error state we have:

$$\mathtt{error}_{n^r_{in}}\mathtt{(\checkmark)} \Leftarrow \mathtt{error}_{m}\mathtt{(\_)}.$$

Above, '_' stands for an anonymous variable. The condition of the implication makes sure that the error state ✓ is only added to the abstract value corresponding to node n^r_{in} if it is reachable.

In order to define $\mathsf{combine}_2{}^\sharp$ we generate the following implication for all variables x other than `ret`:

$$\mathtt{var}_{x,n}\mathtt{(X)} \Leftarrow \mathtt{var}_{x,m}\mathtt{(X)}.$$

For `ret` we define:

$$\texttt{var}_{\texttt{ret},n}\texttt{(X)} \Leftarrow \texttt{var}_{\texttt{ret},n^r_{fi}}\texttt{(X)}\,.$$

Thus, the value of variable `ret` is propagated from the final node of the procedure r to node n, whereas the values of other variables x are propagated from node m. The error state at node n needs to be greater or equal to those at nodes m and n^r_{fi}. Therefore, the error states corresponding to nodes m and n^r_{fi} are joined:

$$\begin{aligned}\texttt{error}_n\texttt{(X)} &\Leftarrow \texttt{error}_m\texttt{(X)}\,.\\ \texttt{error}_n\texttt{(X)} &\Leftarrow \texttt{error}_{n^r_{fi}}\texttt{(X)}\,.\end{aligned}$$

The definitions of the clauses for simulating assignments and Boolean expressions are as in [15] which we repeat here for the sake of self-containedness.

5.2 Horn Clauses for Assignments

First, we discuss the case of assignments, i.e., transformers of edges of the following form:

$$(m, [\![x\texttt{:=}e_1(x_1,\ldots,x_n), y\texttt{:=}e_2(y_1,\ldots,y_m)]\!]^\sharp, n)$$

If no error occurs, then x and y are updated, the values of other variables remain unchanged. Accordingly, for all variables $z \neq x$ and $z \neq y$ the following clauses are defined, which propagate their values unmodified:

$$\texttt{var}_{z,n}\texttt{(X)} \Leftarrow \texttt{var}_{z,m}\texttt{(X)}\,.$$

In order to handle errors, the following implication is generated in addition:

$$\texttt{error}_n\texttt{(X)} \Leftarrow \texttt{error}_m\texttt{(X)}\,. \tag{8}$$

Values of variables on the left hand sides of the assignments are defined by the following clauses.

- For edges with label $(x\texttt{:=\#}, x\texttt{:=\#})$ we have $\texttt{var}_{x,n}\texttt{(\#)} \Leftarrow \texttt{var}_{x,m}\texttt{(\_)}$ where '_' denotes an anonymous logic variable. The implication is required to ensure that `#` is added to the predicate $\texttt{var}_{x,n}$ only if m may be reachable. At a reachable node n for all variables x there is a tree τ so that $\texttt{var}_{x,n}(\tau)$ holds.
- For edges with $(x\texttt{:=}y, x\texttt{:=}y)$ we have: $\texttt{var}_{x,n}\texttt{(X)} \Leftarrow \texttt{var}_{y,m}\texttt{(X)}$.
- For edges with $(x\texttt{:=}\sigma_2\texttt{(}y\texttt{,}z\texttt{)}, x\texttt{:=}\sigma_2\texttt{(}y\texttt{,}z\texttt{)})$ we have:

$$\texttt{var}_{x,n}\texttt{(}\sigma_2\texttt{(L,R))} \Leftarrow \texttt{var}_{y,m}\texttt{(L),var}_{z,m}\texttt{(R)}\,.$$

- For edges with $(x\texttt{:=}y\texttt{/1}, x\texttt{:=}y\texttt{/1})$ we have $\texttt{var}_{x,n}\texttt{(L)} \Leftarrow \texttt{var}_{y,m}\texttt{(}\sigma_2\texttt{(L,\_))}$ for all $\sigma_2 \in \Sigma_2$. As an example, let us suppose that the abstract value of variable `pList` at node m is a model of the predicate `qPlist` according to the implications in (7). Using the command `id := pList/1` we can assign

the head of the list into variable `id`. The implication defining the abstract value of variable `id` after the assignment is:

$$\texttt{var}_{\texttt{id},n}\texttt{(L)} \Leftarrow \texttt{var}_{\texttt{pList},m}\texttt{(patientRecord(L,\_))}.$$

However, during the analysis the label of the root of the tree in a variable needs to be treated as unknown. Therefore, the implication is repeated for all possible binary alphabet elements $\sigma_2 \in \Sigma_2$.
An error is caused by an expression of the form x`/1`, if the content of x does not have children, i.e., it is a leaf. Therefore, in addition the following is defined:

$$\begin{aligned} \texttt{error}_n(\lightning) &\Leftarrow \texttt{var}_{y,m}\texttt{(\#)}. \\ \texttt{error}_n(\lightning) &\Leftarrow \texttt{var}_{y,m}(bv). \\ \texttt{error}_n(\top) &\Leftarrow \texttt{var}_{y,m}\texttt{(}\star\texttt{)}. \end{aligned} \tag{9}$$

- For edges with (x`:=`y`/2`, x`:=`y`/2`) we have $\texttt{var}_{x,n}\texttt{(R)} \Leftarrow \texttt{var}_{y,m}(\sigma_2\texttt{(\_,R)})$ for all $\sigma_2 \in \Sigma_2$. The implications handling the error state are identical to those in (9).
- By edges with (l, l) where $l = x\texttt{:=}\lambda_t\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_k\texttt{)}$, it needs to be examined whether the arguments of the function λ_t contain secret. The implications below are used for the purpose, where the second and third lines are defined for all $\sigma_2 \in \Sigma_2$:

$$\begin{aligned} \texttt{secret(}\star\texttt{)}. \\ \texttt{secret}(\sigma_2\texttt{(L,\_)}) &\Leftarrow \texttt{secret(L)}. \\ \texttt{secret}(\sigma_2\texttt{(\_,R)}) &\Leftarrow \texttt{secret(R)}. \end{aligned}$$

Concerning the resulting value of x we have:

$$\texttt{var}_{x,n}(bv) \Leftarrow \texttt{var}_{x_1,m}\texttt{(\_)},\texttt{var}_{x_2,m}\texttt{(\_)},\ldots,\texttt{var}_{x_k,m}\texttt{(\_)}. \tag{10}$$

$$\texttt{var}_{x,n}\texttt{(}\star\texttt{)} \Leftarrow \texttt{var}_{x_i,m}\texttt{(X)},\texttt{secret(X)},\texttt{var}_{x_1,m}\texttt{(\_)},\ldots,\texttt{var}_{x_k,m}\texttt{(\_)}. \tag{11}$$

According to implication (10), the value of x at node n will potentially be bv if all of the arguments $x_1, ..., x_k$ of λ_t are defined. Furthermore, according to implication (11), if any of the input variables depends on the secret, then the resulting abstract value will also contain $\star$. There is an implication of the form (11) defined for all arguments x_i of λ_t.
- By edges with labels (x`:=`$e(x_1, \ldots, x_k)$, `skip`) or (`skip`, x`:=`$e(x_1, \ldots, x_k)$) we have:

$$\texttt{var}_{x,n}\texttt{(}\star\texttt{)} \Leftarrow \texttt{var}_{x_1,m}\texttt{(\_)},\ldots,\texttt{var}_{x_k,m}\texttt{(\_)}.$$

If the effect of an edge consists of an assignment and a `skip` command, then in the resulting abstract state the value of the variable on the left-hand-side becomes $\star$. This indicates that its value might be different in the corresponding two concrete states. At this point we cannot take the values of variables on the right-hand-side into consideration.
If the expression is of the form x`/1` or x`/2`, then we have in addition: $\texttt{error}_n\texttt{(}\top\texttt{)} \Leftarrow \texttt{var}_{x,m}\texttt{(\_)}$ in order to indicate that an error may occur only in one member of the pair of corresponding concrete states.

5.3 Horn Clauses for Boolean Expressions

Now we discuss abstract transformers with Boolean expressions. In our implementation we treat two branching constructs composable only if their conditional expressions are syntactically equivalent. Therefore, here we only need to treat the corresponding combinations of Boolean expressions.

- By edges labeled (b, b), $(b, \texttt{skip})$ or $(\texttt{skip}, b)$, where $b = \lambda_b(x_1, x_2, \ldots, x_k)$, the values of all variables y occurring in the program are propagated the following way:

$$\texttt{var}_{y,n}\texttt{(X)} \Leftarrow \texttt{var}_{y,m}\texttt{(X)},\texttt{var}_{x_1,m}\texttt{(\_)}, \ldots, \texttt{var}_{x_k,m}\texttt{(\_)}.$$

  In other words, it is checked whether the input variables of the conditional expression have been defined, in order to ensure that the node m is reachable. The actual values of variables are propagated without modification.
- In case the label of the root of a tree is tested using an edge having label of the form $(\texttt{top(}x\texttt{)=}\sigma, \texttt{top(}x\texttt{)=}\sigma)$, then the following clauses are defined to propagate the values of variables $y \neq x$ if $\sigma \in \Sigma_2$:

$$\begin{aligned} \texttt{var}_{y,n}\texttt{(X)} &\Leftarrow \texttt{var}_{y,m}\texttt{(X)},\texttt{var}_{x,m}\texttt{(}\sigma\texttt{(\_,\_))}. \\ \texttt{var}_{y,n}\texttt{(X)} &\Leftarrow \texttt{var}_{y,m}\texttt{(X)},\texttt{var}_{x,m}\texttt{(}\star\texttt{)}. \end{aligned} \tag{12}$$

  The value of the variable x is propagated as well:

$$\texttt{var}_{x,n}\texttt{(}\sigma\texttt{(L,R))} \Leftarrow \texttt{var}_{x,m}\texttt{(}\sigma\texttt{(L,R))}. \tag{13}$$

$$\texttt{var}_{x,n}\texttt{(}\sigma\texttt{(}\star\texttt{,}\star\texttt{))} \Leftarrow \texttt{var}_{x,m}\texttt{(}\star\texttt{)}. \tag{14}$$

  If $\sigma = \texttt{\#}$ then $\sigma\texttt{(}X\texttt{,}Y\texttt{)}$ is exchanged with # in (12), (13) and (14).
- For edges with $(\texttt{top(}x\texttt{)=}\sigma, \texttt{skip})$ or $(\texttt{skip}, \texttt{top(}x\texttt{)=}\sigma)$ (12) needs to be repeated for all variables $y \neq x$, and (13) needs to be repeated in order to propagate the value of x as well. In addition $\texttt{var}_{x,n}\texttt{(}\star\texttt{)} \Leftarrow \texttt{var}_{x,m}\texttt{(}\star\texttt{)}$ needs to be defined.
- By edges having labels of the form $(\neg\texttt{top(}x\texttt{)=}\sigma, \neg\texttt{top(}x\texttt{)=}\sigma)$, $(\neg\texttt{top(}x\texttt{)=}\sigma,$ $\texttt{skip})$ or $(\texttt{skip}, \neg\texttt{top(}x\texttt{)=}\sigma)$, the values of the variables are propagated only in the case, when the root of the value of x is labeled with some $\delta \neq \sigma$. Therefore, the following implication is defined for all variables y other than x and for all alphabet elements $\delta \in \Sigma_2 \setminus \{\sigma\}$:

$$\texttt{var}_{y,n}\texttt{(X)} \Leftarrow \texttt{var}_{y,m}\texttt{(X)},\texttt{var}_{x,m}\texttt{(}\delta\texttt{(\_,\_))}. \tag{15}$$

  In order to handle the value of x as well, the following implication is defined for all $\delta \in \Sigma_2 \setminus \{\sigma\}$:

$$\texttt{var}_{x,n}\texttt{(}\delta\texttt{(L,R))} \Leftarrow \texttt{var}_{x,m}\texttt{(}\delta\texttt{(L,R))}. \tag{16}$$

  Additionally, we need to define (15) and (16) so that $\delta\texttt{(}X\texttt{,}Y\texttt{)}$ is replaced by $\star$, and if $\sigma \neq \texttt{\#}$ then by # too.

– In case the two components of the label of an edge are the negations of each other, e.g., $(\neg b(x_1, x_2, ..., x_k),\ b(x_1, x_2, ..., x_k))$, then the values of variables need to be propagated only in the case, when at least one of the variables in the argument depends on the secret. Assuming that b is a function, the simultaneous execution of the two steps cannot take place otherwise. Accordingly, the following is defined for all variables y occurring in the program and for all variables $x_i \in \{x_1, ..., x_k\}$:

$$\mathtt{var}_{y,n}\mathtt{(X)} \Leftarrow \mathtt{var}_{y,m}\mathtt{(X)},\mathtt{var}_{x_i,m}\mathtt{(Z)},\mathtt{secret(Z)}. \tag{17}$$

The transfer functions of edges labeled with Boolean expressions propagate the error state without modification, therefore we generate implications of the form (8) for each of them.

5.4 Discussion

In [15], we have proven that the abstract transformer $[\![f, g]\!]^\sharp$ for a pair (f, g) as defined by Horn clauses is correct. In other words, if $[\![f]\!](s_0) = s$ and $[\![g]\!]t_0 = t$ where $(s_0, t_0) \in \gamma(D_0)$ and $[\![f, g]\!]^\sharp D_0 = D$, then $(s, t) \in \gamma(D)$ holds. Therefore, noninterference for a particular output variable x holds at program exit n_{fi}, if the predicate $\mathtt{var}_{x,n_{fi}}$ does not accept trees containing $\star$. Algorithmically, therefore, the analysis boils down to computing (or approximating) the least model of the set of Horn clauses defined for the program. Now we observe that the head of each clause possibly generated by our analysis, is of one of the following forms:

$$h \quad ::= \quad \mathtt{p} \mid \mathtt{p(}X\mathtt{)} \mid \mathtt{p(}\sigma_0\mathtt{)} \mid \mathtt{p(}\sigma_2\mathtt{(}X_1\mathtt{,}X_2\mathtt{))},$$

where X_1 and X_2 are distinct. Therefore, all of them belong to the class of Horn clauses $\mathcal{H}_1$ [18,25]. Finite sets of clauses from this class are known to have least models consisting of *regular* sets. Moreover, finite automata characterizing these regular sets can be effectively computed by means of a normalization procedure.

We have carried out the analysis of the self-composition of the program in Listing 6. The initial abstract values of the input variables are defined by the following implications:

$$\begin{gathered}\mathtt{var}_{\mathtt{pList},n_{in}}\mathtt{(X)} \Leftarrow \mathtt{qPlist(X)}.\\ \mathtt{var}_{\mathtt{test},n_{in}}\mathtt{(}\star\mathtt{)}.\ \mathtt{var}_{\mathtt{patientId},n_{in}}\mathtt{(}bv\mathtt{)}.\end{gathered} \tag{18}$$

The initial values of all other variables x are defined by predicates of the form $\mathtt{var}_{x,n_{in}}\mathtt{(\#)}$, the initial error state is defined by $\mathtt{error}_{n_{in}}\mathtt{(\checkmark)}$.

The result of the analysis reveals that the final abstract value of the variable `pList` equals to the initial one defined by the predicate `qPlist` in (18). In other words, the secret remains in the variable `test` and in the health status of the records in the list, but does not interfere with other values. Thus, the call string 0 approach is sufficient to verify noninterference. The reason is that the procedure calls of the original program are all within a similar security context. In applications, where this is not the case, more information about calling contexts may be required.

6 Related Work and Conclusion

Information flow analysis can be traced at least as far back as to the fundamental work of Denning [10] who provided a lattice model for information flow, and Volpano et al. who provided a first sound type system to analyze such properties [24]. Since then various extensions of the original ideas have been proposed. For a still valuable overview see, e.g, in [20]. An alternative approach to information flow analysis is through program dependence graphs as, e.g., realized in the system Joana for Java [12].

Both approaches loose information when the control flow depends on secret data. Information flow properties, on the other hand, can be naturally formalized as 2-hypersafety properties. As observed by Clarkson and Schneider, the verification of 2-hypersafety properties can be reduced to the verification of ordinary safety properties for self-compositions of programs [6]. This direct road to a more precise information flow analysis has been further explored by Barthe et al. [2, 3] and Banerjee et al. [17, 22] who provide proof methods based on self-compositions. In our predecessor paper [15], we have introduced syntactic matching of programs to provide appropriate self-compositions and relational abstract interpretation in order to obtain fully automatic analyses. These ideas have been exemplified by means of an intraprocedural analysis of XML transformers. Our present paper has reviewed that approach and extended it with a generic treatment of procedures. In principle, any technique for interprocedural analysis (e.g., [9,14,21]) can be applied together with suitable abstract domains which allow to express potential differences between reaching pairs of program states. For the case of programs manipulating XML trees, we used public views to identify these differences. Analyzing the sets of potentially reaching public views for 2-programs, turned out to be quite analogous to analyzing the sets of potentially reaching XML values for ordinary programs.

References

1. A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, Dieter Koenig, M. Marin, V. Mehta, S. Thatte, D. Rijn, P. Yendluri, and A. Yiu. Web services business process execution language version 2.0 (OASIS standard). WS-BPEL TC OASIS, `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`, 2007.
2. Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2011.
3. Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2013*, volume 7734 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2013.
4. Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML path language (XPath) 2.0 (second

edition). World Wide Web Consortium, Recommendation REC-xpath20-20101214, 14 December 2010.
5. Niklas Broberg and David Sands. Paralocks: Role-based information flow control and beyond. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 431–444, 2010.
6. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
7. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.
8. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.
9. Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1977.
10. Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
11. Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
12. Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009.
13. Michael Kay. XSL transformations (XSLT) version 2.0. World Wide Web Consortium, Recommendation REC-xslt20-20070123, 23 January 2007.
14. Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction, 4th International Conference on Compiler Construction (CC'92)*, pages 125–140. Springer, LNCS 641, 1992.
15. Máté Kovács, Helmut Seidl, and Bernd Finkbeiner. Relational abstract interpretation for the verification of 2-hypersafety properties. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM Conference on Computer and Communications Security (CCS'13)*, pages 211–222. ACM, 2013.
16. Andrew C. Myers. JFlow: Practical mostly-static information flow control. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241. ACM, 1999.
17. Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Dependent type theory for verification of information flow and access control policies. *ACM Trans. Program. Lang. Syst.*, 35(2):6, 2013.
18. Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Normalizable Horn clauses, strongly recognizable relations, and Spi. In Manuel V. Hermenegildo and Germán Puebla, editors, *Static Analysis, 9th International Symposium, SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2002.
19. Mateusz Pawlik and Nikolaus Augsten. RTED: A robust algorithm for the tree edit distance. *PVLDB*, 5(4):334–345, 2011.
20. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

21. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Application*, pages 189–233. Prentice-Hall, 1981.
22. Gordon Stewart, Anindya Banerjee, and Aleksandar Nanevski. Dependent types for enforcement of information flow and erasure policies in heterogeneous data structures. In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming (PPDP '13)*, pages 145–156. ACM, 2013.
23. Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.
24. Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
25. Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In Harald Ganzinger, editor, *CADE-16, 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 1999.