Relational Abstract Interpretation for the Verification of 2-Hypersafety Properties

Máté Kovács
Technische Universität München, Germany
kovacsm@in.tum.de

Helmut Seidl
Technische Universität München, Germany
seidl@in.tum.de

Bernd Finkbeiner
Universität des Saarlandes, Germany
finkbeiner@cs.uni-sb.de

ABSTRACT

Information flow properties of programs can be formalized as *hyperproperties* specifying the relation of multiple executions. In this paper, we therefore introduce a framework for proving 2-hypersafety properties by means of abstract interpretation. The main idea is to apply abstract interpretation on the self-compositions of the control flow graphs of programs. As a result, our method is inherently capable of analyzing relational properties of even dissimilar programs.

Constructing self-compositions of control flow graphs is nontrivial. Therefore, we present an algorithm for constructing quality self-compositions driven by a tree distance measure between the abstract syntax trees of subprograms. Finally, we demonstrate the applicability of the approach by proving intricate information flow properties of programs written in a simple language for tree manipulation motivated by the Web Services Business Process Execution Language.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

Keywords

abstract interpretation; hyperproperties; information flow control; semi-structured data

1. INTRODUCTION

Abstract interpretation [12] is a well established approach for proving safety properties of programs. However, an interesting class of properties, namely information flow properties, are best formalized not as safety properties, but as *safety hyperproperties* [10]. Safety hyperproperties are not invariants maintained by individual runs of systems but are properties of multiple runs.

In [10] it has been observed that the verification of k-hypersafety properties can be reduced to the verification of ordinary safety properties of the k-fold self-composition of the program. In this paper we apply this idea to control flow graphs (CFG), and present a method for proving 2-hypersafety properties. We apply the general approach for proving noninterference of programs written in an XML manipulating language.

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516721.

Listing 1: A BPEL fragment updating the health status of patients based on the variable `test`

```
<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>
```

Noninterference means that two executions are indistinguishable based on public observations if their initial states only differ in secret data [19]. Consider, e.g., the code fragment in Listing 1 written in the Web Services Business Process Execution Language [3] (BPEL). The fragment is meant to update the health status of patients 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`. 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. It is challenging, because data manipulation is carried out in different branches depending on the secret. By comparing two executions corresponding to the two branches, we notice, however, that only the health status of the patient with the given id can be different in the results, while all public data remain equal. In this paper we show a method to prove properties like this automatically using static analysis.

We regard the execution of a program as the sequence of assignments and condition evaluations that take place during the computation. We take advantage of the fact that inserting `skip` instructions into arbitrary places of an execution does not change the result. Therefore, different *alignments* of a pair of executions can be achieved by the insertion of `skip` instructions. Knowing an initial abstract value describing the potential set of pairs of initial states, we are interested in computing the final abstract value describing the possible set of pairs of states that can result from any pairs of executions. For that we require a relational abstract domain describing *pairs* of concrete states, together with abstract transformers for *pairs* of instructions set in alignment within the two executions. The abstract effect of a given pair of executions w.r.t. a fixed alignment of instructions is obtained by applying the composition of the occurring transformers to the initial abstract value. Since the abstract effect of any alignment results in a safe over-approximation of the desired outcome, we can approximate it by the greatest lower bound over all alignments. In order to obtain a safe approximation for all pairs of executions reaching a given pair of program points, we then take the least upper bound of the values provided for each pair of executions individually. We refer to this value as the *merge over all twin computations* (MTC) solution, and consider it as the ideal solution of the analysis problem.

In general, it might be difficult to compute the MTC solution directly. Instead, we propose to select one promising static alignment of instructions for each pair of executions resulting in a *self-composition* of the CFG of the program. Here, the key problem is to find decent self-compositions of CFGs, which is nontrivial. Therefore, we present an algorithm to construct them recursively based on the abstract syntax trees of programs. In order to achieve maximal precision, the structural similarities of two subprograms are taken into account using a tree distance measure during the construction. Once a particular self-composition of the CFG is ready, an over-approximation of the MTC solution is obtained as a solution to a constraint system formulated based on the self-composition.

Our goal is to apply the general approach described above to verify the security of web service orchestrations implemented e.g., in BPEL. Web service orchestrations may implement workflows within organizations having access to sensitive data, while communicating with external partners over the Internet. Therefore, the security of these processes and the information they maintain should be granted. On the other hand, the central concept of the web service technologies is that data is stored in XML documents. Accordingly, the BPEL language is optimized for XML manipulation using XPath [8] expressions and XSLT [22] transformations. The extra challenge here is that information flow policies may not only refer to values of variables but to parts of the documents stored in variables too. Accordingly, information flow policies are composed in terms of public views of document trees specifying the potential positions of secrets. Therefore, we introduce an abstract domain based on regular sets of public views of documents together with abstract transformers modeling operations on document trees using Horn clauses. During the analysis implications are generated to specify the relations of abstract values corresponding to the nodes of the self-composition of the CFG. These implications fall into a special category, which can be solved e.g., using $\mathcal{H}_1$-normalization [31, 37].

In order to simplify the presentation and concentrate on the key issues of information flow, we restricted ourselves to a minimalistic "assembly" language for tree manipulation.

To summarize, this paper has the following contributions:

- We introduce the ideal solution of the analysis problem of pairs of executions of a program, and propose to overapproximate it by applying abstract interpretation on a particular self-composition of the corresponding CFG.

- A concrete algorithm for the construction of self-compositions of CFGs is proposed, which is driven by a tree distance measure in order to take into account the similarities of subprograms.

- We demonstrate the applicability of the general framework using a complex abstract domain for semi-structured data. Based on the self-composition, a translation is proposed into Horn clauses, which can be solved, e.g., by means of $\mathcal{H}_1$-normalization [31, 37].

- The approach is evaluated on case studies found in the literature [2, 5, 16].

The rest of the paper is structured as follows. In Section 2 the theoretical framework is elaborated. In Section 3 we show a method to construct the necessary self-compositions of CFGs based on structured programs, while in Section 4 we instantiate our approach for the case of a tree-manipulating programming language. In Section 5 we validate the presented analysis using practical experiments on case studies found in the literature. In Section 6 we relate our work to others, and in Section 7 we conclude.

2. MERGE OVER ALL TWIN COMPUTATIONS

In this section we establish the formal groundwork for proving 2-hypersafety properties using relational abstract interpretation.

We define the semantics of programs by means of control-flow graphs (CFG). A CFG is a tuple $G = (N, E, n_{in}, n_{fi})$ consisting of a set of nodes N including the unique initial and final nodes n_{in} and n_{fi}, and a set of directed and labeled edges E. Members of the set E are of the form (n_1, f, n_2), where the nodes n_1 and n_2 are the initial and final nodes of the edge and the label is f. Labels f of edges represent state transformers $[\![f]\!] : S \nrightarrow S$, i.e., partial mappings on the set of states S. We presume the existence of a special label `skip` with $[\![\texttt{skip}]\!]s = s$ for all $s \in S$. A run or execution of a program is defined by a path from the initial node to the final node. The effect of the sequence of labels $\pi = f_1...f_n$ of edges in a run is given by the composition of the effects of the individual labels: $[\![\pi]\!]s_0 = [\![f_1...f_n]\!]s_0 = [\![f_n]\!] \circ ... \circ [\![f_1]\!]s_0$. In the rest, we denote the set of sequences of labels of paths from node n_1 to node n_2 by $n_1 \rightsquigarrow n_2$.

Properties we aim to verify in this paper are defined below:

Definition 1. *We define a 2-hypersafety property by an initial and a final relation of program states $\rho_{in}, \rho_{fi} \subseteq S \times S$. A program given by CFG $G = (N, E, n_{in}, n_{fi})$ satisfies the 2-hypersafety property specified by ρ_{in} and ρ_{fi}, if for all pairs of initial states $(s_0, t_0) \in \rho_{in}$ and all pairs of final states $s = [\![\pi_1]\!]s_0$ and $t = [\![\pi_2]\!]t_0$ reachable by arbitrary computations $\pi_1, \pi_2 \in n_{in} \rightsquigarrow n_{fi}$, it holds that $(s, t) \in \rho_{fi}$.*

When applying abstract interpretation to this problem, we may choose a *Cartesian* abstraction of pairs of states. This means that each program state in the pairs is abstracted separately. Let us briefly argue why this approach may often return unsatisfactory results. Consider, e.g., the program $p =$`if h>5 then l:=l+2 else l:=l+2`, with integer variables `h` and `l`. Assume that ρ_{in} and ρ_{fi} consist of all pairs of states where the values of variable `l` are equal. Since the precise initial values for `l` are unknown, the final values for `l` will also be unknown, implying that their equality cannot be inferred. Therefore, better results can be obtained by using a *relational* abstraction of pairs of program states. A relational abstraction of pairs of program states may record the fact that the variables `l` in the two program states tracked in parallel are equal, and that this equality is preserved by all possible pairs of program executions.

In general, let $(\mathbb{D}, \sqsubseteq)$ be a complete lattice forming a Galois connection $(\mathcal{P}(S\times S), \alpha, \gamma, \mathbb{D})$ with the powerset of pairs of states $\mathcal{P}(S \times S)$, where $\alpha : \mathcal{P}(S \times S) \to \mathbb{D}$ is an abstraction function and $\gamma : \mathbb{D} \to \mathcal{P}(S \times S)$ is a concretization function. The only requirement for abstract transformers $[\![f, g]\!]^\sharp$ of pairs of labels (f, g) (otherwise called *twin steps*) is that they must satisfy the following property:

$$\gamma([\![f,g]\!]^\sharp d) \quad \supseteq \quad \{ \quad \begin{array}{l}(s',t') \mid \exists (s,t) \in \gamma(d) : \\ [\![f]\!]s = s' \wedge [\![g]\!]t = t' \end{array} \quad \} \tag{1}$$

Given two sequences[1], π_1 and π_2, the set of all possible *alignments* $A(\pi_1, \pi_2)$ is recursively defined by:

$$\begin{array}{lcl} A(\varepsilon,\varepsilon) & = & \varepsilon \cup \{(\texttt{skip},\texttt{skip})\omega \mid \omega \in A(\varepsilon,\varepsilon)\} \\ A(\varepsilon, g\pi) & = & \{(\texttt{skip}, g)\omega \mid \omega \in A(\varepsilon,\pi)\} \cup \\ & & \{(\texttt{skip},\texttt{skip})\omega \mid \omega \in A(\varepsilon, g\pi)\} \\ A(f\pi,\varepsilon) & = & \{(f,\texttt{skip})\omega \mid \omega \in A(\pi,\varepsilon)\} \cup \\ & & \{(\texttt{skip},\texttt{skip})\omega \mid \omega \in A(f\pi,\varepsilon)\} \\ A(f\pi_1, g\pi_2) & = & \{(f,g)\omega \mid \omega \in A(\pi_1,\pi_2)\} \cup \\ & & \{(\texttt{skip},g)\omega \mid \omega \in A(f\pi_1,\pi_2)\} \cup \\ & & \{(f,\texttt{skip})\omega \mid \omega \in A(\pi_1,g\pi_2)\} \cup \\ & & \{(\texttt{skip},\texttt{skip})\omega \mid \omega \in A(f\pi_1,g\pi_2)\} \end{array} \tag{2}$$

According to (2), an alignment of two sequences of labels of a CFG, π_1 and π_2, is a sequence of twin steps ω representing both of the original runs. The insertion of `skip` labels does not change the result of a run. Therefore, if $[\![\pi_1]\!]s_0 = s$, $[\![\pi_2]\!]t_0 = t$, $\omega \in A(\pi_1, \pi_2)$ and $\omega = (f_1, g_1)...(f_n, g_n)$, then we can write that $(s,t) = [\![\omega]\!](s_0,t_0) = [\![(f_1, g_1)...(f_n, g_n)]\!](s_0, t_0) = [\![f_n, g_n]\!] \circ ... \circ [\![f_1, g_1]\!](s_0, t_0)$, where for each i, $(s_i, t_i) = [\![f_i, g_i]\!](s_{i-1}, t_{i-1})$ if $s_i = [\![f_i]\!]s_{i-1}$ and $t_i = [\![g_i]\!]t_{i-1}$ so that $s = s_n$ and $t = t_n$.

Given two runs π_1, π_2 and an abstract value d_0, we are interested in the most precise abstract value d that can be computed using the abstract semantics of twin steps:

$$d = \bigsqcap_{\omega \in A(\pi_1,\pi_2)} [\![\omega]\!]^\sharp d_0$$

Since there can be multiple pairs of paths executed on the members of the concretization of an initial abstract value, we obtain a sound overapproximation for the abstract value at any pair of nodes by computing the least upper bound of the abstract effects for all possible pairs of paths of the CFG reaching these nodes.

[1]In this paper we apply the alignment construction to sequences of labels of CFGs, and in later sections to programs, i.e., sequences of commands as well.

Definition 2. *Given a CFG $G = (N, E, n_{in}, n_{fi})$ and the initial abstract value d_0, the* merge over all twin computations solution *is defined by:*

$$MTC(G, d_0) = \bigsqcup_{\substack{\pi_1 \in n_{in} \rightsquigarrow n_{fi} \\ \pi_2 \in n_{in} \rightsquigarrow n_{fi}}} \; \bigsqcap_{\omega \in A(\pi_1,\pi_2)} [\![\omega]\!]^\sharp d_0$$

The MTC solution can be considered as the extension of the *meet over all paths* (MOP) solution of Kam and Ullman [21] to pairs of paths on the CFG.

Theorem 1. *Consider a pair of sequences of labels $\pi_1, \pi_2 \in n_{in} \rightsquigarrow n_{fi}$ on the CFG $G = (N, E, n_{in}, n_{fi})$, and states $s_0, s, t_0, t \in S$, where $s = [\![\pi_1]\!]s_0$, $t = [\![\pi_2]\!]t_0$ and $(s_0, t_0) \in \gamma(d_0)$. In this case $d \sqsupseteq MTC(G, d_0)$ implies $(s, t) \in \gamma(d)$.*

The proof can be found in [23].

According to Theorem 1, the MTC solution is an abstraction of all possible pairs of states resulting from any pair of executions of the program. This result can be used to infer how relations of initial states are translated into relations of final states and thus prove 2-hypersafety properties. In general it might be difficult, though, to compute the MTC solution directly. A perhaps less precise, but still sound solution is obtained by restricting the set of alignments $A(\pi_1, \pi_2)$ for which the greatest lower bound is computed in the MTC solution. In particular, we may even fix a single alignment for each pair of paths. Such a fixed alignment can be obtained by constructing a self-composition GG of the CFG G. For later use, we define more generally, when is a graph a composition of two CFGs.

Definition 3. *Given the CFGs $G = (N^G, E^G, n^G_{in}, n^G_{fi})$ and $H = (N^H, E^H, n^H_{in}, n^H_{fi})$, $GH = (N', E', n'_{in}, n'_{fi})$ is a composition of G and H, if each edge in E' has a label (f, g) where f and g are labels of G and H, respectively, or* `skip`*, and furthermore for all $\pi_G \in n^G_{in} \rightsquigarrow n^G_{fi}$ and $\pi_H \in n^H_{in} \rightsquigarrow n^H_{fi}$ there is an $\omega_{\pi_G,\pi_H} \in n'_{in} \rightsquigarrow n'_{fi}$ so that $\omega_{\pi_G,\pi_H} \in A(\pi_G, \pi_H)$.*

According to Definition 3, a CFG GG is considered to be a self-composition of G, if for all pairs of paths on G there is a path on GG so that the latter is an alignment of the former two paths. Note that due to the insertion of `skip` operations, a self-composition may be quite different from the Cartesian product of the two graphs, where every two aligned paths have exactly the same length.

Theorem 2. *Given the CFG $G = (N, E, n_{in}, n_{fi})$ and a self-composition of it $GG = (N', E', n'_{in}, n'_{fi})$, the following holds for all d_0:*

$$\bigsqcup_{\omega \in n'_{in} \rightsquigarrow n'_{fi}} [\![\omega]\!]^\sharp d_0 \sqsupseteq MTC(G, d_0)$$

For the proof, please, refer to [23].

By Theorem 2, any solution of the analysis problem [21] corresponding to the self composition GG of G is a safe overapproximation of $\mathrm{MTC}(G, d_0)$.

Proving a 2-hypersafety property by means of our methods succeeds in two steps. First, a self-composition GG of the CFG is constructed. The graph GG gives rise to a constraint system over a suitable relational abstract domain, which describes how relations of states are transformed by pairs of edges. A solution to this constraint system then provides the analysis result. Accordingly, the two key practical problems consist in finding a *decent* self-composition and a *decent* abstract domain, which achieve reasonable precision at an acceptable price. Note that different analysis results can be obtained if different self-compositions are chosen. Theorem 2 guarantees that all results are sound, and therefore allow to increase the precision of the analysis.

3. SELF-COMPOSITIONS OF CONTROL FLOW GRAPHS

When an appropriate alignment of computations is chosen, we assume that abstract transformers $[\![f, f']\!]^\sharp$ for pairs of identical or similar actions f, f' are more precise than abstract transformers for arbitrary pairs. Therefore, the goal is to maximize the number of edges labeled (f, f') in a self-composition of a CFG. If the program in question has been specified by means of a structured programming language, a quality self-composition can be obtained by means of syntactically matching abstract syntax trees (ASTs) of program fragments.

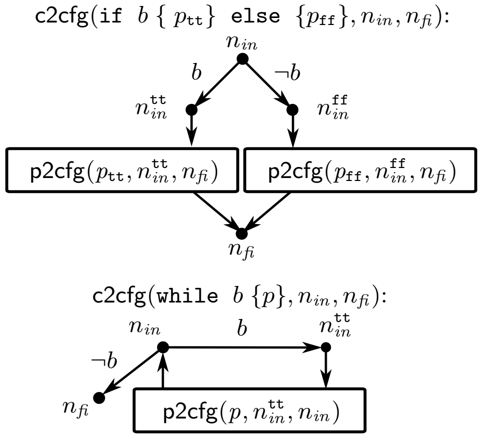

c2cfg(if b { $p_{\mathtt{tt}}$} else {$p_{\mathtt{ff}}$}, n_{in}, n_{fi}):
n_{in}
b
$\neg b$
$n_{in}^{\mathtt{tt}}$
$n_{in}^{\mathtt{ff}}$
p2cfg($p_{\mathtt{tt}}, n_{in}^{\mathtt{tt}}, n_{fi}$)
p2cfg($p_{\mathtt{ff}}, n_{in}^{\mathtt{ff}}, n_{fi}$)
n_{fi}
c2cfg(while b {p}, n_{in}, n_{fi}):
n_{in}
b
$n_{in}^{\mathtt{tt}}$
$\neg b$
n_{fi}
p2cfg($p, n_{in}^{\mathtt{tt}}, n_{in}$)

Figure 1: CFGs corresponding to branching constructs

In order to illustrate the approach, assume that programs are generated by the nonterminal p of the grammar below:

$$\begin{array}{llll} \text{(program)} & p & ::= & c; \mid c;p \\ \text{(command)} & c & ::= & \mathtt{skip} \mid x \mathrel{\mathtt{:=}} e \mid \mathtt{while}\ b\ \{p\} \mid \\ & & & \mathtt{if}\ b\ \{p_{\mathtt{tt}}\}\ \mathtt{else}\ \{p_{\mathtt{ff}}\} \end{array} \quad (3)$$

The grammar (3) defines a structured programming language, where the state of execution and the syntax of expressions e and b will be instantiated for our tree-manipulating language in Section 4.

CFGs of programs are constructed by two mutually recursive functions "program to CFG" (p2cfg) and "command to CFG" (c2cfg). The CFG corresponding to the program p can be obtained by means of the function $\mathsf{p2cfg}(p, n_{in}, n_{fi})$, where the initial and final nodes of the resulting graph are n_{in} and n_{fi}, respectively. In case $p = c_1;...;c_m$, then $m-1$ fresh nodes are instantiated, and $\mathsf{c2cfg}(c_i, n_{i-1}, n_i)$ is called for each i to construct the CFG fragment corresponding to the command c_i so that $n_0 = n_{in}$ and $n_m = n_{fi}$. $\mathsf{c2cfg}(\mathtt{skip}, n_i, n_{i+1}) = (n_i, \mathtt{skip}, n_{i+1})$ and $\mathsf{c2cfg}(x\mathtt{:=}e, n_i, n_{i+1}) = (n_i, x\mathtt{:=}e, n_{i+1})$, furthermore, the CFG fragments corresponding to branching constructs are shown in Figure 1.

A self-composition of the CFG corresponding to a program can be constructed by two mutually recursive functions "pair of programs to CFG" (pp2cfg) and "pair of commands to CFG" (pc2cfg). The function $\mathsf{pp2cfg}(p_1, p_2, n_{in}, n_{fi})$ can also be called for two different programs, p_1 and p_2, in order to align the two alternatives in a conditional. The function makes use of a distance measure for trees as introduced, e.g., in [34]. The evaluation of $\mathsf{pp2cfg}(p_1, p_2, n_{in}, n_{fi})$ proceeds in two steps:

Step 1: Computing a Best Alignment of Two Programs.

Here, we adapt the definition of sets of alignments (2) to two sequences of commands. An optimal alignment of two sequences of commands, $c_1;...;c_k$ and $d_1;...;d_l$, with respect to a tree distance measure td is the sequence of pairs of commands ω_{opt}, where the distances $\mathsf{td}(c_i^{\omega_{\text{opt}}}, d_i^{\omega_{\text{opt}}})$ of the ASTs of the aligned pairs of commands $(c_i^{\omega_{\text{opt}}}, d_i^{\omega_{\text{opt}}})$ is minimal. Formally, an optimal alignment ω_{opt} of $p_1 = c_1;...;c_k$ and $p_2 = d_1;...;d_l$ is defined as:

$$\omega_{\text{opt}} = \underset{\omega \in A(p_1,p_2)}{\arg\min} \sum_{1 \le i \le |\omega|} \mathsf{td}(\omega[i].1, \omega[i].2) \quad (4)$$

In (4), $\omega[i]$ stands for the pair of commands number i in the sequence ω, and $\omega[i].1$ and $\omega[i].2$ denote the first and second components of the pair respectively. In our implementation we use the *Robust Tree Edit Distance* described in [32] as tree distance measure.

Step 2: Computing the Compositions of CFGs of Pairs of Commands.

If the alignment is ready, the compositions of the CFGs corresponding to the aligned pairs of commands need to be constructed. Assume that the chosen alignment of the two programs is $\omega = (c_1, d_1)...(c_k, d_k)$. In this case we instantiate a fresh node for each $1 \le i \le k-1$, and call $\mathsf{pc2cfg}(c_i, d_i, n_{i-1}, n_i)$ so that $n_0 = n_{in}$ and $n_k = n_{fi}$.

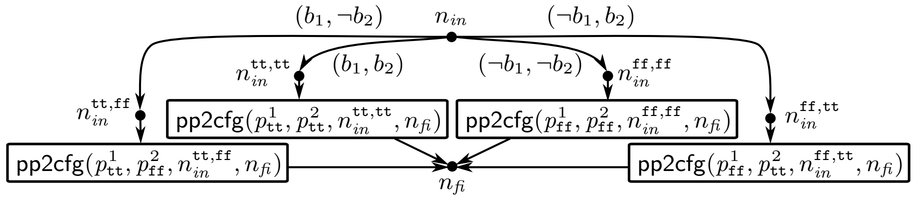
Now we discuss how the subgraphs corresponding to pairs of commands are constructed. First, the roots of the abstract syntax trees corresponding to the commands are compared. The roots of the ASTs corresponding to c and d are considered *composable* in the following cases:

- $c = d = x \mathrel{\mathtt{:=}} e$ or $c = d = \mathtt{skip}$, i.e., in case of assignments the commands need to be identical including the variable on the left and the expression on the right.
- $c = \mathtt{if}\ b_1\ \{p_{\mathtt{tt}}^1\}\ \mathtt{else}\ \{p_{\mathtt{ff}}^1\}$ and $d = \mathtt{if}\ b_2\ \{p_{\mathtt{tt}}^2\}\ \mathtt{else}\ \{p_{\mathtt{ff}}^2\}$.
- $c = \mathtt{while}\ b_1\ \{p_1\}$ and $d = \mathtt{while}\ b_2\ \{p_2\}$.

In case the roots of the ASTs of the commands c and d are not composable, then we put them in sequence. This is achieved by means of two additional functions. The function $\mathsf{skip1}(G)$ replaces each label f of the edges of the graph G with $(\mathtt{skip}, f)$, and similarly, $\mathsf{skip2}(G)$ replaces these labels with $(f, \mathtt{skip})$. In order to compute $\mathsf{pc2cfg}(c, d, n_{in}, n_{fi})$ in this case, we instantiate a fresh node n' and compute $\mathsf{skip2}(\mathsf{c2cfg}(c, n_{in}, n'))$ and $\mathsf{skip1}(\mathsf{c2cfg}(d, n', n_{fi}))$.

It remains to consider a pair of commands c and d where the roots of the corresponding ASTs are composable. If they are not branching constructs, then $\mathsf{pc2cfg}(c, d, n_{in}, n_{fi}) = (n_{in}, (c, d), n_{fi})$.

Now consider a pair of branching constructs. The corresponding self-compositions are illustrated in Figure 2. Here, The resulting CFG of $\mathsf{pc2cfg}(p_1, p_2, n_{in}, n_{fi})$, where $p_1 = \texttt{if}\ \ b_1\ \{\ p^1_{\texttt{tt}}\}\ \texttt{else}\ \{p^1_{\texttt{ff}}\}$, and $p_2 = \texttt{if}\ \ b_2\ \{\ p^2_{\texttt{tt}}\}\ \texttt{else}\ \{p^2_{\texttt{ff}}\}$:


$(b_1, \neg b_2)$
n_{in}
$(\neg b_1, b_2)$
$n^{\texttt{tt},\texttt{tt}}_{in}$
(b_1, b_2)
$(\neg b_1, \neg b_2)$
$n^{\texttt{ff},\texttt{ff}}_{in}$
$n^{\texttt{tt},\texttt{ff}}_{in}$
$\mathsf{pp2cfg}(p^1_{\texttt{tt}}, p^2_{\texttt{tt}}, n^{\texttt{tt},\texttt{tt}}_{in}, n_{fi})$
$\mathsf{pp2cfg}(p^1_{\texttt{ff}}, p^2_{\texttt{ff}}, n^{\texttt{ff},\texttt{ff}}_{in}, n_{fi})$
$n^{\texttt{ff},\texttt{tt}}_{in}$
$\mathsf{pp2cfg}(p^1_{\texttt{tt}}, p^2_{\texttt{ff}}, n^{\texttt{tt},\texttt{ff}}_{in}, n_{fi})$
n_{fi}
$\mathsf{pp2cfg}(p^1_{\texttt{ff}}, p^2_{\texttt{tt}}, n^{\texttt{ff},\texttt{tt}}_{in}, n_{fi})$

The resulting CFG of $\mathsf{pc2cfg}(\texttt{while}\ \ b_1\ \{\ p_1\}, \texttt{while}\ \ b_2\ \{\ p_2\}, n_{in}, n_{fi})$:

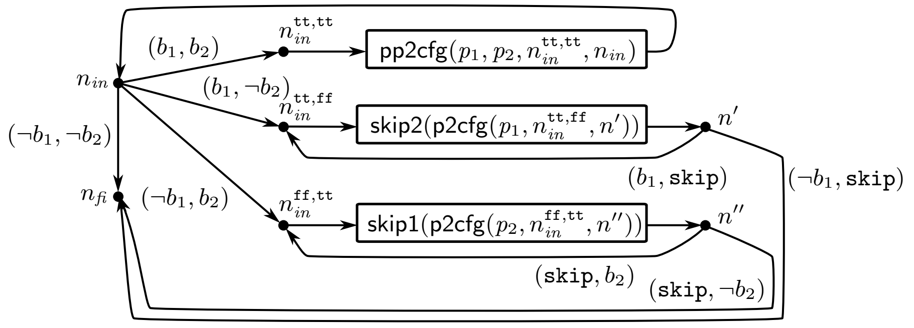

$n^{\texttt{tt},\texttt{tt}}_{in}$
(b_1, b_2)
$\mathsf{pp2cfg}(p_1, p_2, n^{\texttt{tt},\texttt{tt}}_{in}, n_{in})$
n_{in}
$(b_1, \neg b_2)$
$n^{\texttt{tt},\texttt{ff}}_{in}$
$(\neg b_1, \neg b_2)$
$\mathsf{skip2}(\mathsf{p2cfg}(p_1, n^{\texttt{tt},\texttt{ff}}_{in}, n'))$
n'
$(b_1, \texttt{skip})$
$(\neg b_1, \texttt{skip})$
n_{fi}
$(\neg b_1, b_2)$
$n^{\texttt{ff},\texttt{tt}}_{in}$
$\mathsf{skip1}(\mathsf{p2cfg}(p_2, n^{\texttt{ff},\texttt{tt}}_{in}, n''))$
n''
$(\texttt{skip}, b_2)$
$(\texttt{skip}, \neg b_2)$

Figure 2: The compositions of CFGs corresponding to branching constructs

the idea is to compose the CFGs of the bodies of the branching constructs according to all possible valuations of the conditional expressions, and then connect them with edges so that they become compositions of the branching constructs according to Definition 3. In particular, the composition of two loops results in three. One loop handles the case when the bodies are executed simultaneously, the other two execute the body of only one original loop. This way we handle the situation, when the two original loops execute a different number of times.

Note that the functions pp2cfg and pc2cfg are mutually recursive. Accordingly, subprograms at each level of the ASTs are aligned separately.

The following theorem states the correctness of the construction of self-compositions of CFGs introduced above.

THEOREM 3. *Consider a program p together with its CFG G constructed by the function $\mathsf{p2cfg}(p, n_{in}, n_{fi})$. In this case, the resulting CFG of the function $\mathsf{pp2cfg}(p, p, n'_{in}, n'_{fi})$ is a self-composition of G according to Definition 3.*

The proof is by induction on the abstract syntax tree of p. Please, refer to [23] for the details.

4. PROVING NONINTERFERENCE

In this section we show how to apply the developments of Sections 2 and 3 in order to verify information flow properties of tree-manipulating programs. In Section 4.1 the programming language (3) is instantiated with an expression language optimized for tree manipulation, and in Section 4.2 an abstract interpretation is introduced. In Section 4.3 a case study is presented, where we model the BPEL fragment of Listing 1 with a program implemented in our language, and prove an information flow property on it.

4.1 A Language for Tree Manipulation

XML documents are unranked ordered trees, where there is no bound on the number of children of nodes. For the sake of simplicity, here we represent tree-shaped data using binary trees by means of the first-child-next-sibling (FCNS) encoding [11]. Figure 3 illustrates an unranked tree-shaped record corresponding to a patient in a database on the top, and on the bottom its binary representation as a member of a list. As Figure 3 shows, the FCNS encoding maps the first child of a node in an unranked tree to its first child, and its sibling to the right to its second child. The set of binary trees over the alphabet Σ_2 of symbols for binary nodes and Σ_0 for nullary nodes is denoted by $\mathcal{T}_{\Sigma_2,\Sigma_0}$. Binary nodes labeled with the elements of Σ_2 model XML tags, while nullary nodes labeled with the elements of Σ_0 model textual entities of XML documents. Therefore, we identify the latter by putting their values between ' characters. One can think of the labels of nullary nodes as untyped values represented as strings. The extra label $\texttt{\#} \in \Sigma_0$ denotes the empty forest.

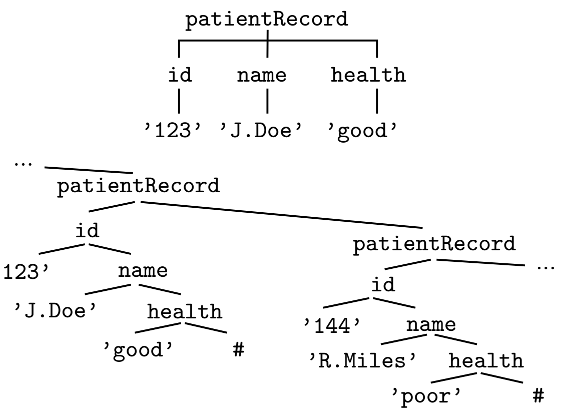


Figure 3: A document tree describing a patient (top), and the corresponding binary representation as member of a list (bottom)

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

In (5), the expression language of the programming language of (3) is defined. Tree expressions e evaluate to new tree-shaped values based on already existing ones, and Boolean expressions b compute Boolean values. The values of x/1 and x/2 are the first and second subtrees of the value stored in x, respectively, given that the root of x is binary. Otherwise, if x is a leaf, they result in the error state $\lightning$. The result of $\sigma_2(x,y)$ is a tree having root labeled $\sigma_2 \in \Sigma_2$ with the content of x as its first and the content of y as its second child. In order to allow for arbitrary computations, we have introduced the interpreted functions λ_t and λ_b returning textual (otherwise called basic) and Boolean values respectively. In the following examples we will use infix operators like less or equal to `=<` or equivalence `=` as instances for interpreted function symbols with the intuitive semantics.

$$
\begin{array}{rcl}
[\![\texttt{top(}x\texttt{)=}\sigma]\!]s & = & s \text{ if } \sigma \in \Sigma_2 \cup \{\texttt{\#}\} \text{ and} \\
 & & s(x) \text{ has root labeled with } \sigma \\
[\![\neg\texttt{top(}x\texttt{)=}\sigma]\!]s & = & s \text{ if } \sigma \in \Sigma_2 \cup \{\texttt{\#}\} \text{ and} \\
 & & s(x) \text{ has root labeled with } \sigma' \neq \sigma \\
[\![\lambda_b\texttt{(}x_1\texttt{,}x_2\texttt{,}...\texttt{)}]\!]s & = & s \text{ if } [\![\lambda_b]\!](s(x_1), s(x_2), ...) \text{ holds}
\end{array}
$$

$$
[\![x\texttt{:=}y]\!]s = s[x \mapsto s(y)] \qquad [\![x\texttt{:=\#}]\!]s = s[x \mapsto \texttt{\#}]
$$
$$
[\![x\texttt{:=}\sigma_2\texttt{(}x_1\texttt{,}x_2\texttt{)}]\!]s = s[x \mapsto \sigma_2(s(x_1), s(x_2))]
$$
$$
[\![x\texttt{:=}y\texttt{/1}]\!]s = \begin{cases} s[x \mapsto t_1] & \text{if } s(y) = \sigma_2\texttt{(}t_1\texttt{,}t_2\texttt{)} \text{ for some} \\ & \text{label } \sigma_2\text{, and trees } t_1 \text{ and } t_2 \\ \lightning & \text{otherwise} \end{cases}
$$
$$
[\![x\texttt{:=}y\texttt{/2}]\!]s = \begin{cases} s[x \mapsto t_2] & \text{if } s(y) = \sigma_2\texttt{(}t_1\texttt{,}t_2\texttt{)} \text{ for some} \\ & \text{label } \sigma_2\text{, and trees } t_1 \text{ and } t_2 \\ \lightning & \text{otherwise} \end{cases}
$$
$$
[\![x\texttt{:=}\lambda_t\texttt{(}x_1\texttt{,}x_2\texttt{,}...\texttt{)}]\!]s = s[x \mapsto [\![\lambda_t]\!](s(x_1), s(x_2), ...)] \text{ or } \lightning
$$
$$
[\![f]\!]\lightning = \lightning \text{ for all edges } f
$$

Figure 4: State transformers of edges

The semantics of a program is given by means of the corresponding CFG according to Section 2. Here, a program state s is a mapping $(Var \rightarrow \mathcal{T}_{\Sigma_2,\Sigma_0}) \cup \{\lightning\}$ from the set of variables to binary trees, or the error state $\lightning$. The state transformers of edges are defined in Figure 4, where $s[x \mapsto v]$ denotes a state with each variable $y \neq x$ having value $s(y)$, and $s(x) = v$. In particular, we define $[\![f]\!]\lightning = \lightning$ for all edges.

4.2 Proving Noninterference Using Relational Abstract Interpretation

By the specification of information flow policies [15] we distinguish between two secrecy levels, namely *public* or *low* (L) and *secret* or *high* (H). The meaning is that the principals not entitled to learn information classified H, should not be able to make inferences on them based on observing data classified L. We are only concerned with termination *insensitive* noninterference, i.e., we neglect the amount of secret that can be learned by observing the non-termination of executions. A program is deemed secure only if for all pairs of states s_0 and t_0 the following holds. If the public parts of s_0 and t_0 are equal, i.e., $s_0|_L = t_0|_L$, then for all pairs of executions starting at the initial node of the CFG of the program in states s_0 and t_0, and reaching the final node in states s and t, respectively, the public parts of the two resulting states are equal as well, i.e., $s|_L = t|_L$.

In order to verify an information flow policy, our analysis maintains abstract states $d : Var \rightarrow \mathcal{P}(\mathcal{T}_{\Sigma_2,\{\texttt{\#},bv,\star\}})$, which are mappings from variables to sets of trees possibly containing occurrences of dedicated leaves bv and $\star$. A pair of states (s,t) is in the concretization[2] $(s,t) \in \gamma(d)$ of d, if for all variables x, $(s(x), t(x)) \in \gamma(d(x))$ holds. A pair of trees τ_1, τ_2 is in the concretization of a set Λ of abstract trees, if Λ contains a tree τ such that both τ_1 and τ_2 can be obtained from τ by replacing the occurrences of bv with identical basic values, and the occurrences of $\star$ with any, possibly different subtrees. Consider, e.g., the language $\Lambda = \{\texttt{a(}\star\texttt{,}bv\texttt{)}, \texttt{b(}bv\texttt{,}\star\texttt{)}\}$. Then $(\tau_1, \tau_2) \in \gamma(\Lambda)$ for $\tau_1 = $ `a('Top secret!','42')`, and $\tau_2 = $ `a('Top secret, too!', '42')`. Accordingly, if $(\tau_1, \tau_2) \in \gamma(\Lambda)$, then the elements of Λ can be considered as potential public views of the trees τ_1 and τ_2, where the occurrences of $\star$ identify the positions of secrets. A public view is a piece of relational information on two trees in the sense that it determines their common upper part and the locations of potential secrets. Therefore, an information flow policy that can be verified by our method needs to be given in the form of a set of public views for each variable in the initial abstract value of the analysis. Abstract values do not record precise information on basic values. Leaves labeled bv only mark the positions of basic values having secrecy level L.

In order to deal with program errors as well, we extend this basic approach by recording in the abstract state for a program point not only public views of variables, but additionally provide Boolean values $B_\lightning$ and $B_\star$. If $B_\lightning$ holds, then $(\lightning, \lightning)$ can also be a member of the concretization. If $B_\star$ holds, then either of the two components of a pair of concrete states may be $\lightning$ meaning that the occurrence of an error may depend on the secret.

Given an initial abstract value d_0 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 pair of jointly reachable program points. In our analysis, sets of public views for variables x occurring at a node n are described by means of unary predicates $\texttt{var}_{x,n}$, which are defined by means of Horn clauses. Formally, $\tau \in d(x)$ at node n if $\texttt{var}_{x,n}(\tau)$ holds. The values of $B_\lightning$ and $B_\star$ at node n are represented by means of the nullary predicates $\texttt{public\_error}_n$ and $\texttt{secret\_error}_n$ respectively.

There are two kinds of Horn clauses. A first group is used for specifying the information flow policy in terms of a set of initial public views for each variable at the initial node n_{in}. A second group describes how the views at different program points are related to each other. These clauses are obtained from the self-composition of the CFG.

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

[2] We overload the notation γ by applying it to the concretization of abstract states as well as to the concretization of abstract sets of trees.

tree automata here, but note that their languages can be defined by means of clauses of the form:

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

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

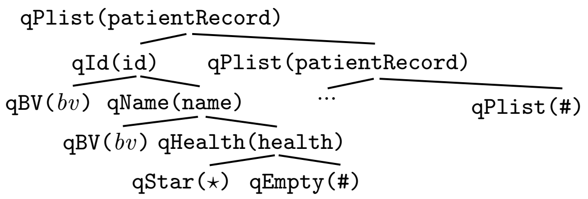


Figure 5: A run of the automaton specified in (6) accepting the public view of a list of records like that on the bottom of Figure 3 with state qPlist

In our running example, the health status of patients in the list of records seen in Figure 3 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 (6):

$$\begin{array}{rcl} \texttt{qPlist(patientRecord(L,R))} & \Leftarrow & \texttt{qId(L),qPlist(R).} \\ & \texttt{qPlist(\#).} & \\ \texttt{qId(id(L,R))} & \Leftarrow & \texttt{qBV(L),qName(R).} \\ \texttt{qName(name(L,R))} & \Leftarrow & \texttt{qBV(L),qHealth(R).} \\ \texttt{qHealth(health(L,R))} & \Leftarrow & \texttt{qStar(L),qEmpty(R).} \\ \multicolumn{3}{c}{\texttt{qBV(}bv\texttt{).} \quad \texttt{qStar(}\star\texttt{).} \quad \texttt{qEmpty(\#).}} \end{array} \tag{6}$$

As Figure 5 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 only trees, the first child of which 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} and for this variable is defined by: $\texttt{var}_{\texttt{pList},n_{in}}\texttt{(X)} \Leftarrow \texttt{qPlist(X)}$.

In the following, we describe how the abstract state transformers $[\![f,g]\!]^\sharp$ of edges $(n,(f,g),n')$ 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. Recall that due to our alignment procedure edges either refer to assignments or to Boolean expressions, but never to both.

4.2.1 Assignments as Horn Clauses

First, we discuss the case of assignments, i.e., transformers of the form $[\![x\texttt{:=}e_1(x_1,\dots,x_n), y\texttt{:=}e_2(y_1,\dots,y_m)]\!]^\sharp$. If no error occurs, then x and y are updated, the values of other variables remain unchanged. Error propagation is discussed later in Section 4.2.3. Accordingly, for all variables $z \neq x$ and $z \neq y$ the following clauses are defined, which propagate their values unmodified:

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

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 $\texttt{var}_{x,n'}\texttt{(\#)} \Leftarrow \texttt{var}_{y,n}\texttt{(\_)}$ for all variables y occurring in the program, where '_' denotes an anonymous logic variable. The implication is required to ensure that # is added to the predicate $\texttt{var}_{x,n'}$ only if n may be reachable. We consider a node n as unreachable, if there is no variable x and tree τ so that $\texttt{var}_{x,n}(\tau)$ holds.

- For edges with $(x\texttt{:=}y, x\texttt{:=}y)$ we have: $\texttt{var}_{x,n'}\texttt{(X)} \Leftarrow \texttt{var}_{y,n}\texttt{(X)}$.

- For edges with $(x\texttt{:=}\sigma_2\texttt{(}y\texttt{,}z\texttt{)}, x\texttt{:=}\sigma_2\texttt{(}y\texttt{,}z\texttt{)})$ we have:

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

- For edges with $(x\texttt{:=}y\texttt{/1}, x\texttt{:=}y\texttt{/1})$ we have $\texttt{var}_{x,n'}\texttt{(L)} \Leftarrow \texttt{var}_{y,n}\texttt{(}\sigma_2\texttt{(L,\_))}$ for all $\sigma_2 \in \Sigma_2$. As an example, let us suppose that the abstract value of variable pList at node n is a model of the predicate qPlist according to implications in (6). 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 $\texttt{var}_{\texttt{id},n'}\texttt{(L)} \Leftarrow \texttt{var}_{\texttt{pList},n}\texttt{(patientRecord(L,\_))}$. 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 $x\texttt{/1}$, if the content of x does not have children, i.e., it is a leaf. Therefore, in addition the following is defined:

$$\begin{array}{rcl} \texttt{public\_error}_{n'} & \Leftarrow & \texttt{var}_{y,n}\texttt{(\#).} \\ \texttt{public\_error}_{n'} & \Leftarrow & \texttt{var}_{y,n}\texttt{(}bv\texttt{).} \\ \texttt{secret\_error}_{n'} & \Leftarrow & \texttt{var}_{y,n}\texttt{(}\star\texttt{).} \end{array} \tag{7}$$

- For edges with $(x\texttt{:=}y\texttt{/2}, x\texttt{:=}y\texttt{/2})$ we have $\texttt{var}_{x,n'}\texttt{(R)} \Leftarrow \texttt{var}_{y,n}\texttt{(}\sigma_2\texttt{(\_,R))}$ for all $\sigma_2 \in \Sigma_2$. The implications handling errors are identical to those in (7).

- By edges with (f,f) where $f = x\texttt{:=}\lambda_t\texttt{(}x_1\texttt{,}\dots\texttt{,}x_k\texttt{)}$, 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 $\sigma_2 \in \Sigma_2$:

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

  Concerning the resulting value of x we have:

$$\begin{array}{rcl} \texttt{var}_{x,n'}\texttt{(}bv\texttt{)} & \Leftarrow & \texttt{var}_{x_1,n}\texttt{(\_),}\texttt{var}_{x_2,n}\texttt{(\_),} \\ & & \dots\texttt{,}\texttt{var}_{x_k,n}\texttt{(\_).} \end{array} \tag{8}$$

$$\begin{array}{rcl} \texttt{var}_{x,n'}\texttt{(}\star\texttt{)} & \Leftarrow & \texttt{var}_{x_i,n}\texttt{(X),}\texttt{secret(X),} \\ & & \texttt{var}_{x_1,n}\texttt{(\_),}\dots\texttt{,} \\ & & \texttt{var}_{x_k,n}\texttt{(\_).} \end{array} \tag{9}$$

  According to implication (8), the value of x at node n' will potentially be bv if all of the arguments $x_1,\dots,x_k$ of λ_t are defined. Furthermore, according to implication (9), 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 (9) defined for all arguments x_i of λ_t.

- By edges of the form $(x\texttt{:=}e(x_1,\ldots,x_k),\texttt{skip})$ and $(\texttt{skip},x\texttt{:=}e(x_1,\ldots,x_k))$ we have:

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

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 $\star$ indicating that its value might be different in the corresponding two concrete states. This happens independently of the values of arguments of expressions on the right hand side. If the expression is of the form x/1 or x/2, then we have in addition $\texttt{secret\_error}_{n'} \Leftarrow \texttt{var}_{x,n}(\_)$ in order to indicate that an error may occur only in one member of the pair of corresponding concrete states.

4.2.2 Boolean Expressions as Horn Clauses

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 in addition to the conditions discussed in Section 3.

- 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:

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

In other words, it is checked whether the input variables of the conditional expression have been defined, in order to ensure that the node n 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 $\sigma\in\Sigma_2$:

$$\begin{array}{lcl}\texttt{var}_{y,n'}(\texttt{X}) & \Leftarrow & \texttt{var}_{y,n}(\texttt{X}),\\ & & \texttt{var}_{x,n}(\sigma(\_,\_)).\\ \texttt{var}_{y,n'}(\texttt{X}) & \Leftarrow & \texttt{var}_{y,n}(\texttt{X}),\texttt{var}_{x,n}(\star).\end{array} \tag{10}$$

The value of the variable x is propagated as well:

$$\texttt{var}_{x,n'}(\sigma(\texttt{L,R})) \Leftarrow \texttt{var}_{x,n}(\sigma(\texttt{L,R})). \tag{11}$$

$$\texttt{var}_{x,n'}(\sigma(\star,\star)) \Leftarrow \texttt{var}_{x,n}(\star). \tag{12}$$

If $\sigma=\texttt{\#}$ then $\sigma(X,Y)$ is exchanged with # (10), (11) and (12).

- For edges with $(\texttt{top}(x)\texttt{=}\sigma,\texttt{skip})$ or $(\texttt{skip},\texttt{top}(x)\texttt{=}\sigma)$ (10) and (11) need to be repeated, and in addition $\texttt{var}_{x,n'}(\star) \Leftarrow \texttt{var}_{x,n}(\star)$ defined.

- By edges with $(\neg\texttt{top}(x)\texttt{=}\sigma,\neg\texttt{top}(x)\texttt{=}\sigma)$, $(\neg\texttt{top}(x)\texttt{=}\sigma,\texttt{skip})$ 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 $\delta\in\Sigma_2\setminus\{\sigma\}$:

$$\texttt{var}_{y,n'}(\texttt{X}) \Leftarrow \texttt{var}_{y,n}(\texttt{X}),\texttt{var}_{x,n}(\delta(\_,\_)). \tag{13}$$

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

$$\texttt{var}_{x,n'}(\delta(\texttt{L,R})) \Leftarrow \texttt{var}_{x,n}(\delta(\texttt{L,R})). \tag{14}$$

Additionally, we need to define (13) and (14) so that $\delta(X,Y)$ 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., $(\neg b(x_1,x_2,\ldots,x_k)$, $b(x_1,x_2,\ldots,x_k))$, 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 $x_i\in\{x_1,\ldots,x_k\}$:

$$\begin{array}{lcl}\texttt{var}_{y,n'}(\texttt{X}) & \Leftarrow & \texttt{var}_{y,n}(\texttt{X}),\texttt{var}_{x_i,n}(\texttt{X}i),\\ & & \texttt{secret}(\texttt{X}i).\end{array} \tag{15}$$

4.2.3 Propagating the Error

Finally, in order to propagate the error state, for all edges we define:

$$\begin{array}{lcl}\texttt{public\_error}_{n'} & \Leftarrow & \texttt{public\_error}_{n}.\\ \texttt{secret\_error}_{n'} & \Leftarrow & \texttt{secret\_error}_{n}.\end{array}$$

4.2.4 Discussion

THEOREM 4. *The abstract transformer $[\![f,g]\!]^\sharp$ for a pair (f,g) as defined by Horn clauses is a correct abstract transformer, i.e., it satisfies the conditions of (1). 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.*

PROOF. The statement follows by a comparison of the concrete transformers in Section 4.1 and the implications defined for pairs of labels of edges. □

Because of Theorem 4, the least solution of the set of Horn clauses defined for a program over-approximates the MTC solution. Therefore, for example, noninterference for a particular output variable x holds at program exit n_{fi}, if the predicate $\texttt{var}_{x,n_{fi}}$ does not accept trees containing $\star$.

Algorithmically, therefore, the analysis boils down to computing (or approximating) the model of the set of Horn clauses defined for the program. The head of each clause possibly generated by our analysis is of one of the forms

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

where X_1,X_2 are distinct. Therefore, all of them belong to the class of Horn clauses $\mathcal{H}_1$ [31, 37]. Finite sets of clauses of this class are known to have least models consisting of regular sets. Moreover, finite automata characterizing these regular sets can be effectively computed.

4.3 Case Study

Let us come back to our running example, the routine manipulating the hospital database, which processes the list of records of patients. An implementation of the BPEL code of Listing 1 in our tree-manipulating core language is given by Listing 2:

Listing 2: A routine updating the health status of patients based on blood tests. Subroutines are syntactic sugar, and they are inlined before analysis or execution. Some analysis results are displayed in comments, where nodes of the self-composition of the corresponding CFG are identified by pairs of numbers of lines in the code.

```
   subroutine query {
    found := 'ff';  // $\mathtt{var}_{\mathtt{found},(3,3)}(bv)$.
    while found = 'ff' {
     id := pList/1; // $\mathtt{var}_{\mathtt{id},(5,5)}(\mathtt{X}) \Leftarrow \mathtt{qId(X)}$.
 5   idVal := id/1; // $\mathtt{var}_{\mathtt{idVal},(6,6)}(\mathtt{X}) \Leftarrow \mathtt{qBV(X)}$.
     if idVal = patientId {
      found := 'tt' // $\mathtt{var}_{\mathtt{found},(8,8)}(\mathtt{X}) \Leftarrow \mathtt{qBV(X)}$.
      name := id/2; // $\mathtt{var}_{\mathtt{name},(9,9)}(\mathtt{X}) \Leftarrow \mathtt{qName(X)}$.
      nameVal := name/1;
10              // $\mathtt{var}_{\mathtt{nameVal},(11,11)}(\mathtt{X}) \Leftarrow \mathtt{qBV(X)}$.
      pList := pList/2;
                // $\mathtt{var}_{\mathtt{pList},(3,3)}(\mathtt{X}) \Leftarrow \mathtt{qPlist(X)}$.
     } else {
      pListRev := patientRecord(id,pListRev);
15              // $\mathtt{var}_{\mathtt{pListRev},(16,16)}(\mathtt{X}) \Leftarrow \mathtt{qPlist(X)}$.
      pList := pList/2;
                // $\mathtt{var}_{\mathtt{pList},(3,3)}(\mathtt{X}) \Leftarrow \mathtt{qPlist(X)}$.
   }}}
   subroutine restore {
20  health := health(healthVal,empty);
    name := name(nameVal,health);
                // $\mathtt{var}_{\mathtt{name},(23,23)}(\mathtt{X}) \Leftarrow \mathtt{qName(X)}$.
    id := id(idVal,name);
                // $\mathtt{var}_{\mathtt{id},(25,25)}(\mathtt{X}) \Leftarrow \mathtt{qId(X)}$.
25  pList := patientRecord(id,pList);
                // $\mathtt{var}_{\mathtt{pList},(27,27)}(\mathtt{X}) \Leftarrow \mathtt{qPlist(X)}$.
    while top(pListRev) = patientRecord {
     id := pListRev/1;
                // $\mathtt{var}_{\mathtt{id},(30,30)}(\mathtt{X}) \Leftarrow \mathtt{qId(X)}$.
30   pList := patientRecord(id,pList);
     pListRev := pListRev/2;
                // $\mathtt{var}_{\mathtt{pListRev},(27,27)}(\mathtt{X}) \Leftarrow \mathtt{qPlist(X)}$.
   }}
   // Entry point:
35 empty := #;             // $\mathtt{var}_{\mathtt{empty},(36,36)}(\#)$.
   pListRev := empty;      // $\mathtt{var}_{\mathtt{pListRev},(37,37)}(\#)$.
   if test =< '0.5' {
    call query;
    healthVal  := 'good'; // $\mathtt{var}_{\mathtt{healthVal},(40,44)}(\star)$.
40  call restore;
   } else {
    call query;
    healthVal  := 'poor'; // $\mathtt{var}_{\mathtt{healthVal},(40,44)}(\star)$.
    call restore;
45 }
```

Subroutines, which are not part of the core language, can be considered as abbreviations. Their definitions are meant to be inlined at their call sites before execution or analysis. In the initial state the variable `pList` contains the database, a list of records in the form of a binary tree as illustrated in Figure 3. The result of the blood test is in variable `test` and the patient's identifier to whom the test belongs is provided in `patientId`. The updated list is again stored in the variable `pList` at the end of the execution. The entry point of the program is at line 35. At line 37, a branch is selected based on the blood test. In both branches, the record of the patient is searched for in the database, and the corresponding health status is updated. Finally, the list representing the database is restored. The database is queried in a loop running as long as the record for the patient with identifier `patientId` is not found. If the actual record at the head of the list in variable `pList` is not the one we are looking for, then it is appended to the list in `pListRev`, which stores a prefix of the list of `pList` in reverse order. On the other hand, if the head of `pList` is the record we seek, then the name of the patient is stored in the variable `nameVal`. The restoration of the database succeeds in two steps. First, beginning with line 20 the binary tree representing the record of the actual patient is reconstructed, and appended to the database in `pList` at line 25. In the second step, the records stored in `pListRev` are appended to `pList` using the loop at line 27.

In the self-composition of the CFG corresponding to the program at Listing 2 after subroutines have been inlined, the `query` and `restore` routines of the two branches of the `if` construct at line 37 are aligned with each other. Therefore, the self-composition simulates the simultaneous query and database restoration in both branches. Listing 2 illustrates the result of the analysis on selected places with predicates of the form $\mathtt{var}_{x,(l_1,l_2)}$ in comments, where x is the variable the value of which the predicate describes, and (l_1, l_2) identifies a node in the self-composition corresponding to the pair of nodes l_1 and l_2 of the original CFG of the program. For the sake of clarity, the identifiers of the initial nodes of subgraphs (e.g., those in Figure 1) corresponding to commands coincide with the line numbers of the commands in the program text.

The initial values of variables are given by:

$$\begin{gathered}\mathtt{var}_{\mathtt{pList},(35,35)}(\mathtt{X}) \Leftarrow \mathtt{qPlist(X)}.\\ \mathtt{var}_{\mathtt{test},(35,35)}(\star). \qquad \mathtt{var}_{\mathtt{patientId},(35,35)}(bv).\end{gathered} \tag{16}$$

Accordingly, after executing the instruction at line 4, the content of the variable `id` will be overapproximated by the set accepted by the predicate `qId`. The reason is that the first child of a tree accepted by predicate `qPlist` is accepted by predicate `qId` according to the implications in (6). The sets abstracting the values of variables at any pair of program points can be established with a similar reasoning. In particular, the abstract value of variable `healthVal` at the node corresponding to lines 40 and 44 is $\star$, indicating that it carries confidential information.

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 (16). 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.

5. PRACTICAL EXPERIMENTS

We have implemented a prototype to carry out the analysis described in this paper. We have implemented the function $\mathsf{pp2cfg}(p_1, p_2, n_{in}, n_{fi})$ in OCaml, where the *Robust Tree Edit Distance* algorithm [32] is used to compute the distance between the ASTs of commands. The set of Horn clauses generated from the self-compositions of CFGs is solved using the $\mathcal{H}_1$ solver of [31]. We have carried out the analysis on four examples. *a*) **Blood Test.** This program is our example described in Section 4. *b*) **Joining Tables.** This program implements the join of two tables as published in [16] and in [5]. *c*) **Bayes Classifier.** This program is an implementation of the second example in [16], which demonstrates how to prove the noninterference of a data-mining algorithm, the *Naive Bayes classifier*. *d*) **Medical Records.** This ex-

Experiment	# of lines	$\|G\|$	$\|GG\|$	# of implications	running time	peak memory usage
Joining Tables	26	45	237	3510	2.791 sec	19.17 MiB
Blood Test	86	136	1102	23971	11.152 sec	26.71 MiB
Medical Records	97	163	859	22532	11.794 sec	21.82 MiB
Bayes Classifier	133	241	1561	89720	44.470 sec	89.85 MiB

Table 1: Summary of the runtime behavior of the analysis

ample implements the benchmark of [2], with four different operations on patient data.

For all four examples, our algorithm succeeded to infer noninterference automatically. Table 1 summarizes some quantitative details of our experiments. The experiments have been carried out in a 32 bit virtual machine on a laptop having a 2 GHz Intel$^{®}$ Core^{TM} i7 processor. To each experiment we list the following information: a) Number of lines of the code without comment and empty lines. b) Size of the CFG $|G|$ as the sum of the number of nodes and number of edges. c) Size of the self-composition of the CFG $|GG|$. d) Number of implications generated. e) The complete running time of the analysis including the construction of the self-composition of the CFG, and the computation of models of predicates. f) The peak memory consumption during the verification process involving the computation of the self-composition of the CFG and models of predicates.

For all examples, the self-composition increases the size of the CFG by a factor significantly less then 10. The number of Horn clauses, on the other hand, does not only depend on the size of the corresponding self-composed graph, but additionally also on the number of variables to be tracked. As expected, the running times typically grow with the number of generated clauses.

6. RELATED WORK

This paper relies on the observation [7, 10, 14] that the verification of k-hypersafety properties of programs can be reduced to the verification of safety properties on the k-fold self-compositions of the programs.

Several authors have used self-compositions of programs for the analysis of hyperproperties, i.e., [7,14,30,35]. In [35] a type-system based transformation is presented to construct the self-composition. The self-composition of branching constructs results in one branching construct with the same condition expression. In [30] the same idea is applied for object-oriented languages. Differently from our approach, these solutions cannot take advantage of similarities in the code of different branches. Their motivation is to apply already existing techniques of proving safety properties for the purpose of proving hypersafety properties.

Barthe et al. [5] offer a variety of possibly conditional rewritings of the original program to achieve appropriate alignments. Such conditions may enforce, e.g., that two loops are iterated equally often. These conditions later must be discarded by the theorem prover or remain and then restrict the admissible input values. Furthermore, no algorithm or heuristics is provided how these rules should be applied. In our solution no extra conditions are imposed. Instead, `skip` instructions are inserted where needed. In case the conditions of two different loops are equal, we are still able to infer equal numbers of iterations and thus achieve a perfect alignment. In [6] notions of compositions of CFGs similar to ours are applied for proving relational properties of potentially dissimilar programs with the help of theorem provers. However, no algorithm is provided for the construction of the self-compositions. Furthermore, similarly to [6], our approach can also to deal with dissimilar programs, this capability is taken advantage of when different branches of a branching construct are analyzed.

While our method is based on self-compositions of control flow graphs, other people apply type systems for the verification of noninterference properties, e.g., [9, 27, 28, 36]. An other alternative is to use program dependency graphs [20]. Generally, though, the above solutions consider the program counter to have a security level implying that all data possibly manipulated in secret-dependent branches are secret. Accordingly, they would have difficulties with our running example, the data-base of patients.

In [18] a non-relational abstract interpretation has been presented, where the security of programs is investigated depending on the observational capabilities of attackers. If the public input values are handled as constants in the initial state, then no information leaks to public variables in the final state, if the property that the attackers can observe can be proved to be constant. A similar idea is applied in the runtime information flow monitor described in [24], where constant propagation is used to compute public values for the final states of branching constructs. A different way of applying abstract interpretation for proving the information flow security of programs is presented in [39] and in [4], where the security labels [15] are treated as abstractions of values, and a consistent labeling is computed according to the abstract semantics. The approaches of [4, 15, 18, 24, 39] do not consider self-compositions of the programs, and therefore cannot take advantage of similarities in different branches of branching constructs.

Abstract interpretation of tree-manipulating programs is described, e.g., in [25], where Møller et al. summarize their experiences using XML Graphs as an abstract domain. Our analysis is similar to that, in the sense that the trees generated by the grammars or automata correspond to public views. Each public view, though, represents a relational piece of information, namely, the common part of two tree-structured values.

There are also papers focusing on proving hyperproperties of programs manipulating complex data structures. In [24, 33] runtime monitors have been introduced for the enforcement of information flow properties on programs manipulating tree-structured data. In [5, 6, 29] the authors present potentially interactive approaches using theorem provers, which are also capable of dealing with our examples. In contrast to these techniques, our static analysis based on abstract interpretation is fully automatic.

Since XML is a standard format for hierarchically organizing and communicating data, security here is a major concern. Several authors therefore, have investigated how to enforce access control policies on XML documents by computing "user views", i.e., fragments of the documents accessible for certain users [1, 13, 17, 26]. These approaches however, do not consider self-compositions. Still, it remains for future work to clarify the precise relationship between our methods for verifying noninterference and such enforcement of access control.

7. CONCLUSION AND FUTURE WORK

We have presented a general approach for analyzing 2-hypersafety properties using relational abstract interpretation on self-compositions of control flow graphs. Properties like that naturally occur in information flow security analyses. Our approach is based on appropriate self-compositions of control flow graphs. Here, we presented an algorithm to construct quality self-compositions, which is completely deterministic and well tailored for the application of relational abstract interpretation. In particular, our solution uses the Robust Tree Edit Distance [32] measure in order to align similar program fragments with each other. We have applied the approach to analyze information flow properties of programs manipulating semi-structured data.

As the general idea of self-compositions of CFGs is independent of the programming language and semantics, our framework can be applied to other programming languages as well. Our results open many directions for further research, e.g.: a) In our analysis, different program variables are treated separately. Extra precision, perhaps may be obtained by using predicates that relate the contents of different variables in states. The resulting clauses then may no longer be members of the class $\mathcal{H}_1$. Still, one may apply a Horn clause based approach, but use a first order theorem prover such as, e.g., SPASS [38] as a back-end, instead of an $\mathcal{H}_1$ solver. b) The language we have considered in this paper does not support procedure calls. It still remains an open question how the Horn clause formulation of the analysis can be optimally extended also to recursive procedures.

Acknowledgments

This work was partially supported by the German Research Foundation (DFG) under the project SpAGAT in the priority program 1496 "Reliably Secure Software Systems – RS3".

8. REFERENCES

[1] R. Abassi, F. Jacquemard, M. Rusinowitch, and S. G. El Fatmi. XML access control: from XACML to annotated schemas. In *Second International Conference on Communications and Networking (ComNet)*, pages 1–8, Tozeur, Tunisie, 2010. IEEE Computer Society Press.

[2] R. Accorsi and A. Lehmann. Automatic information flow analysis of business process models. In A. P. Barros, A. Gal, and E. Kindler, editors, *Business Process Management - 10th International Conference, BPM 2012*, volume 7481 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2012.

[3] 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, D. 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.

[4] R. Barbuti, C. Bernardeschi, and N. D. Francesco. Checking security of Java bytecode by abstract interpretation. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC)*, pages 229–236. ACM, 2002.

[5] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In M. Butler and W. Schulte, editors, *17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2011.

[6] G. Barthe, J. M. Crespo, and C. Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In S. N. Artëmov and A. Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2013*, volume 7734 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2013.

[7] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)*, pages 100–114. IEEE Computer Society, 2004.

[8] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0 (second edition). World Wide Web Consortium, Recommendation REC-xpath20-20101214, 14 December 2010.

[9] N. Broberg and D. Sands. Paralocks: role-based information flow control and beyond. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 431–444. ACM, 2010.

[10] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[11] 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.

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[13] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Trans. Inf. Syst. Secur.*, 5(2):169–202, 2002.

[14] Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing, Second International Conference, SPC 2005*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer, 2005.

[15] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[16] G. Dufay, A. P. Felty, and S. Matwin. Privacy-sensitive information flow with JML. In R. Nieuwenhuis, editor, *CADE-20, 20th International Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2005.

[17] I. Fundulaki and M. Marx. Specifying access control policies for XML documents with XPath. In T. Jaeger and E. Ferrari, editors, *SACMAT 2004, 9th ACM Symposium on Access Control Models and Technologies*, pages 61–69. ACM, 2004.

[18] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004*, pages 186–197. ACM, 2004.

[19] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[20] C. Hammer and G. 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.

[21] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, September 1977.

[22] M. Kay. XSL transformations (XSLT) version 2.0. World Wide Web Consortium, Recommendation REC-xslt20-20070123, 23 January 2007.

[23] M. Kovács. Relational abstract interpretation for the verification of 2-hypersafety properties (proofs). Technical Report TUM-I1340, Technische Universität München, Institut für Informatik, Aug. 2013.

[24] M. Kovács and H. Seidl. Runtime enforcement of information flow security in tree manipulating processes. In G. Barthe, B. Livshits, and R. Scandariato, editors, *Engineering Secure Software and Systems - 4th International Symposium, ESSoS 2012*, volume 7159 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2012.

[25] A. Møller and M. I. Schwartzbach. XML graphs in program analysis. *Sci. Comput. Program.*, 76(6):492–515, 2011.

[26] M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML access control using static analysis. *ACM Trans. Inf. Syst. Secur.*, 9(3):292–324, 2006.

[27] A. C. Myers. JFlow: Practical mostly-static information flow control. In A. W. Appel and A. Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241. ACM, 1999.

[28] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 129–142, New York, NY, USA, 1997. ACM Press.

[29] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, pages 165–179. IEEE Computer Society, 2011.

[30] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In D. Gollmann, J. Meier, and A. Sabelfeld, editors, *ESORICS 2006, 11th European Symposium on Research in Computer Security*, volume 4189 of *Lecture Notes in Computer Science*, pages 279–296. Springer, 2006.

[31] F. Nielson, H. R. Nielson, and H. Seidl. Normalizable Horn clauses, strongly recognizable relations, and Spi. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis, 9th International Symposium, SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2002.

[32] M. Pawlik and N. Augsten. RTED: A robust algorithm for the tree edit distance. *PVLDB*, 5(4):334–345, 2011.

[33] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In M. Backes and P. Ning, editors, *ESORICS 2009, 14th European Symposium on Research in Computer Security*, volume 5789 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2009.

[34] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, 1979.

[35] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.

[36] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[37] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *CADE-16, 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 1999.

[38] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In R. A. Schmidt, editor, *CADE-22, 22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.

[39] M. Zanotti. Security typings by abstract interpretation. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis, 9th International Symposium, SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 360–375. Springer, 2002.