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.

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

```xml
<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 ex-
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 expressions and XSLT 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 $H_1$-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 $H_1$-normalization 31,37.
- The approach is evaluated on case studies found in the literature 2,15.

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 $G = (N, E, n_0, n_f)$ consisting of a set of nodes $N$ including the unique initial and final nodes $n_0$ and $n_f$, 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 $[f] : S \rightarrow S$, i.e., partial mappings on the set of states $S$. We presume the existence of a special label skip with $[\text{skip}] s = s$ 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 $\pi = f_1 \ldots f_n$ of edges in a run is given by the composition of the effects of the individual labels: $[\pi] s_0 = (f_1 \ldots f_n) s_0 = [f_n] \circ \ldots \circ [f_1] s_0$. 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 $\rho_0, \rho_f \subseteq S \times S$. A program given by CFG $G = (N, E, n_0, n_f)$ satisfies the 2-hypersafety property specified by $\rho_0$ and $\rho_f$, if for all pairs of initial states $(s_0, t_0) \in \rho_0$, and all pairs of final states $s = [\pi_1] s_0$ and $t = [\pi_2] t_0$ reachable by arbitrary computations $\pi_1, \pi_2 \in n_0 \rightsquigarrow n_f$, it holds that $(s, t) \in \rho_f$. 


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

\[ p = \text{if } h > 5 \text{ then } 1 = 1 \text{ else } 1 = 2, \]

with integer variables \( h \) and 1. Assume that \( \rho_0 \) and \( \rho_1 \) consist of all pairs of states where the values of variable 1 are equal. Since the precise initial values for 1 are unknown, the final values for 1 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 1 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 \( (\mathcal{D}, \sqsubseteq) \) be a complete lattice forming a Galois connection \( (\mathcal{P}(S \times S), \alpha, \gamma, \mathcal{D}) \) with the powerset of pairs of states \( \mathcal{P}(S \times S) \), where \( \alpha : \mathcal{P}(S \times S) \to \mathcal{D} \) is an abstraction function and \( \gamma : \mathcal{D} \to \mathcal{P}(S \times S) \) is a concretization function. The only requirement for abstract transformers \( [f,g]\) of pairs of labels \( (f,g) \) (otherwise called twin steps) is that they must satisfy the following property:

\[ \gamma([f,g]d) \supseteq \{(s',t') \mid (s,t) \in \gamma(d) : [f]s = s' \land [g]t = t'\} \tag{1} \]

Given two sequences of labels, \( \pi_1 \) and \( \pi_2 \), the set of all possible alignments \( A(\pi_1, \pi_2) \) is recursively defined by:

\[
A(\varepsilon, \varepsilon) = \varepsilon \cup \{(\text{skip},\text{skip}) \mid \omega \in A(\varepsilon, \varepsilon)\}
\]
\[
A(\varepsilon, \pi) = \{(\text{skip}, \omega) \mid A(\varepsilon, \pi_1) \cup \{(\text{skip}, \omega) \mid \omega \in A(\varepsilon, \pi)\}\}
\]
\[
A(\pi, \varepsilon) = \{(\omega, \text{skip}) \mid A(\pi, \pi_1) \cup \{(\omega, \text{skip}) \mid \omega \in A(\pi, \varepsilon)\}\}\]
\[
A(\pi_1, \pi_2) = \{(\omega, \omega) \mid A(\pi_1, \pi_2) \cup \{(\omega, \omega) \mid \omega \in A(\pi_1, \pi_2)\}\}
\]

According to Definition 2, 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 \text{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 CFGs \( G = (N,E,n_{in},n_{in},n_{in},n_{in}) \) and \( H = (N',E',n_{in},n_{in},n_{in},n_{in}) \), GH \( = (N',E',n_{in},n_{in},n_{in},n_{in}) \) is a composition of G and H, if each edge in E' has a label \((f,g)\) such that \( f \) and \( g \) are labels of G and H, respectively, or \text{skip}, and furthermore for all \( \pi G \in n_{in}^G \sqsubseteq n_{in}^G \) and \( \pi H \in n_{in}^H \sqsubseteq n_{in}^H \) there is an \( \omega_{\pi G,\pi H} \in n_{in}^G \sqsubseteq n_{in}^H \) such that \( \omega_{\pi G,\pi H} \in A(GH, \pi G, \pi H) \).

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

\[ MTC(G, d_0) = \bigcup_{\pi_1 \in A(n_{in} \sqsubseteq n_{in}) \omega \in A(\pi_1, \pi_2)} \{[\omega]^d_0 \} \]

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 \( \pi_1, \pi_2 \in n_{in} \sqsubseteq n_{in} \) on the CFG \( G = (N,E,n_{in},n_{in}) \), and states \( s_0, s_1, t, \in S \), where \( s = [\pi_1]s_0, t = [\pi_2]t_0 \) and \( (s_0, t_0) \in \gamma(d_0) \). In this case \( d \supseteq MTC(G, d_0) \) implies \((s, t) \in \gamma(d)\).

The proof can be found in 

For the proof, please refer to [23].

**Definition 3.** Given the CFGs \( G = (N^G,E^G,n^G_{in},n^G_{in}) \) and \( H = (N^H,E^H,n^H_{in},n^H_{in}) \), \( GH = (N',E',n_{in},n_{in},n_{in},n_{in}) \) 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 \text{skip}, and furthermore for all \( \pi G \in n_{in}^G \sqsubseteq n_{in}^G \) and \( \pi H \in n_{in}^H \sqsubseteq n_{in}^H \) there is an \( \omega_{\pi G,\pi H} \in n_{in}^G \sqsubseteq n_{in}^H \) so that \( \omega_{\pi G,\pi H} \in A(GH, \pi G, \pi H) \).

According to Definition 2, 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 \text{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_{in}) and a self-composition of it GG = (N', E', n_{in}, n_{in}) the following holds for all \( d_0 \):

\[ \bigcup \{[\omega]^d_0 \} \supseteq MTC(G, d_0) \]

By Theorem 2, any solution of the analysis problem [21] corresponding to the self composition GG of G is a safe overapproximation of 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 \( [f, f']^p \) 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.

![CFGs corresponding to branching constructs](image)

**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:

\[
\begin{align*}
\text{(program)} & \::= c_1 | c_1 ; c_2 \\
\text{(command)} & \::= \text{skip} | x := e | \text{while } b \{ p \} | \text{if } b \{ p \} \text{ else } \{ p \}
\end{align*}
\]

The grammar 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 \( p2cfg(p, n_{in}, n_{f}) \), where the initial and final nodes of the resulting graph are \( n_{in} \) and \( n_{f} \), respectively. In case \( p = c_1 ; \ldots ; c_m \), then \( m-1 \) fresh nodes are instantiated, and \( c2cfg(c_i, n_{i-1}, n_i) \) is called for each \( i \) to construct the CFG fragment corresponding to the command \( c_i \), so that \( n_0 = n_{in} \) and \( n_{m} = n_{f} \). c2cfg(\text{skip}, n_{in}, n_{f+1}) = (n_{in}, \text{skip}, n_{f+1}) and \( c2cfg(x := e_i, n_{in}, n_{f+1}) = (n_{in}, x := e_i, n_{f+1}) \), 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 \( pp2cfg(p_1, p_2, n_{in}, n_{f}) \) 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 \( pp2cfg(p_1, p_2, n_{in}, n_{f}) \) 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, \( c_1 ; \ldots ; c_k \) and \( d_1 ; \ldots ; d_l \), with respect to a tree distance measure \( \omega \) is the sequence of pairs of commands \( \omega_{opt} \), where the distances \( \omega_{opt}(c_i, d_j) \) of the ASTs of the aligned pairs of commands \( (c_i, d_j) \) is minimal. Formally, an optimal alignment \( \omega_{opt} \) of \( p_1 = c_1 ; \ldots ; c_k \) and \( p_2 = d_1 ; \ldots ; d_l \) is defined as:

\[
\omega_{opt} = \arg \min_{\omega \in \Lambda(p_1, p_2)} \sum_{1 \leq i \leq |\omega|} \omega(i, 1, \omega(i, 2))
\]

In [4, 5], \( \omega[i] \) stands for the pair of commands number \( i \) in the sequence \( \omega \). \( \omega[i, 1] \) 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 \( \omega = (c_1, d_1) ; \ldots ; (c_k, d_k) \). In this case we instantiate a fresh node for each \( 1 \leq i \leq k-1 \), and call \( pc2cfg(c_i, d_i, n_{in-1}, n_i) \) so that \( n_0 = n_{in} \) and \( n_k = n_{f} \).

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:

\[
\begin{align*}
& c = d = x := e \text{ or } c = d = \text{skip}, \text{ i.e., in case of assignments the commands need to be identical including the variable on the left and the expression on the right.} \\
& c = \text{if } b_1 \{ p_1 \} \text{ else } \{ p_2 \} \text{ and } d = \text{if } b_2 \{ p_2 \} \text{ else } \{ p_1 \}.
\end{align*}
\]

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 \( \text{skip1}(G) \) replaces each label \( f \) of the edges of the graph \( G \) with \( \text{skip} \), and similarly, \( \text{skip2}(G) \) replaces these labels with \( (f, \text{skip}) \). In order to compute \( pc2cfg(c, d, n_{in}, n_{f}) \) in this case, we instantiate a fresh node \( n' \) and compute \( \text{skip2}(\text{c2cfg}(c, n_{in}, n')) \) and \( \text{skip1}(\text{c2cfg}(d, n', n_f)) \).

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 \( \text{pc2cfg}(c, d, n_{in}, n_{f}) = (n_{in}, c, d, n_{f}) \).

Now consider a pair of branching constructs. The corresponding self-compositions are illustrated in Figure 2. Here,
The resulting CFG of $\text{pc2cfg}(p_1, p_2, n_m, n_f)$, where $p_1 = \text{if } b_1 \{ p_{1t} \} \text{ else } \{ p_{1f} \}$, and $p_2 = \text{if } b_2 \{ p_{2t} \} \text{ else } \{ p_{2f} \}$:

\[
\begin{align*}
(b_1, -b_2) & \quad n_m \\
(b_1, b_2) & \quad (b_1, -b_2) \\
(p_{1t}, -b_1) & \quad -b_1, b_2 \\
(-b_1, -b_2) & \quad n_f
\end{align*}
\]

The resulting CFG of $\text{pc2cfg}(\text{while } b_1 \{ p_1 \}, \text{while } b_2 \{ p_2 \}, n_m, n_f)$:

\[
\begin{align*}
(b_1, b_2) & \quad n_m \\
(b_1, -b_2) & \quad (b_1, -b_2) \\
(p_{1t}, -b_1) & \quad -b_1, b_2 \\
(-b_1, -b_2) & \quad n_f
\end{align*}
\]

The resulting CFG of $\text{pp2cfg}(p_1, p_2, n_m, n_f)$:

\[
\begin{align*}
(b_1, b_2) & \quad n_m \\
(b_1, -b_2) & \quad (b_1, -b_2) \\
(p_{1t}, -b_1) & \quad -b_1, b_2 \\
(-b_1, -b_2) & \quad n_f
\end{align*}
\]

\[
\begin{align*}
\text{Theorem 3.} & \quad \text{Consider a program } p \text{ together with its CFG } G \text{ constructed by the function } p2cfg(p, n_m, n_f). \text{ In this case, the resulting CFG of the function } pp2cfg(p, n_m, n_f) \text{ is a self-composition of } G \text{ according to Definition 3.}
\end{align*}
\]

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 4.2 an abstract interpretation and in Section 4.3 an abstract interpretation is introduced. In Section 4.4 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 member of a list (bottom).

Figure 3: A document tree describing a patient (top), and the corresponding binary representation as member of a list (bottom)
(tree expressions) \[ e ::= \# | x | x/1 | x/2 | \sigma_2(x,y) | \lambda_1(x_1,x_2,...) \] (Boolean expressions) \[ b ::= \top(x)=\sigma | \lambda_0(x_1,x_2,...) \] \[ (5) \]

In [5], the expression language of the programming language of [3] is defined. Tree expressions evaluate to new tree-shaped values based on already existing ones, and Boolean expressions 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 \(\varepsilon\).

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 \(\lambda_1\) and \(\lambda_0\) returning textual (otherwise called basic) and Boolean values respectively. In the following examples we will use infix operators like less or equal to \(\leq\) or equivalence \(=\) as instances for interpreted function symbols with the intuitive semantics:

\[
\begin{align*}
\llbracket \top(x)=\sigma \rrbracket s &= s \text{ if } \sigma \in \Sigma_2 \cup \{\#\} \text{ and } s(x) \text{ has root labeled with } \sigma \\
\llbracket \neg \top(x)=\sigma \rrbracket s &= s \text{ if } \sigma \in \Sigma_2 \cup \{\#\} \text{ and } s(x) \text{ has root labeled with } \sigma' \neq \sigma \\
\llbracket \lambda_0(x_1,x_2,...) \rrbracket s &= s \text{ if } \lambda_0((s(x_1),s(x_2),...)) \text{ holds} \\
\llbracket x:=y \rrbracket s &= s[x \mapsto y](s) \quad \llbracket x:=\# \rrbracket s = s[x \mapsto \#] \\
\llbracket x:=\sigma_2(x_1,x_2) \rrbracket s &= s[x \mapsto \sigma_2(s(x_1),s(x_2))] \\
\llbracket x:=y/1 \rrbracket s &= \begin{cases} 
\text{label } \sigma_2, \text{ and trees } t_1 \text{ and } t_2 \\
\varepsilon \text{ otherwise} 
\end{cases} \\
\llbracket x:=y/2 \rrbracket s &= \begin{cases} 
\text{label } \sigma_2, \text{ and trees } t_1 \text{ and } t_2 \\
\varepsilon \text{ otherwise} 
\end{cases} \\
\llbracket x:=\lambda_1(x_1,x_2,...) \rrbracket s &= s[x \mapsto [\lambda_1((s(x_1),s(x_2),...))] \text{ or } \varepsilon \\
\llbracket f \rrbracket \varepsilon &= \varepsilon \text{ for all edges } f 
\end{align*}
\]

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 \((\text{Var} \rightarrow \mathcal{T}_{\Sigma_2,\Sigma_0}) \cup \{\varepsilon\}\) from the set of variables to binary trees, or the error state \(\varepsilon\). 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 \(\llbracket f \rrbracket \varepsilon = \varepsilon\) for all edges.

4.2 Proving Noninterference Using Relational Abstract Interpretation

By the specification of information flow policies [12], we distinguish between two secrecy levels, namely \textit{public} or low \((L)\) and \textit{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 \(s_0\) the following holds. If the public parts of \(s_0\) and \(s_0\) are equal, i.e., \(s_0|_L = s_0|_L\), then for all pairs of executions starting at the initial node of the CFG of the program in states \(s_0\) and \(s_0\), and reaching the final node in states \(s\) and \(s\), respectively, the public parts of the two resulting states are equal as well, i.e., \(s|_L = s|_L\).

In order to verify an information flow policy, our analysis maintains abstract states \(d : \text{Var} \rightarrow \mathcal{P}(\mathcal{T}_{\Sigma_2,\Sigma_0}(\bullet,\mathbf{b},\mathbf{*}))\), which are mappings from variables to sets of trees possibly containing occurrences of dedicated leaves \(b\mathbf{v}\) and \(\mathbf{*}\). A pair of states \((s,t)\) is in the concretization \(\llbracket s,t \rrbracket \in \gamma(d)\) of \(d\), if for all variables \(x\), \((s(x),t(x)) \in \gamma(d(x))\) holds. A pair of trees \(s_1,t_2\) is in the concretization of a set \(\Lambda\) of abstract trees, if \(\Lambda\) contains a tree \(t\) such that both \(s_1\) and \(t_2\) can be obtained from \(t\) by replacing the occurrences of \(b\mathbf{v}\) with identical basic values, and the occurrences of \(\mathbf{*}\) with any, possibly different subtrees. Consider, e.g., the language \(\Lambda = \{a(*,b\mathbf{v}),b(b\mathbf{v},\mathbf{*)}\}\). Then \((s_1,t_2) \in \gamma(\Lambda)\) for \(s_1 = a(\text{'Top secret'},'42')\), and \(t_2 = a(\text{'Top secret', tool!}', '42')\). Accordingly, if \((s_1,t_2) \in \gamma(\Lambda)\), then the elements of \(\Lambda\) can be considered as potential views of the trees \(s_1\) and \(t_2\), where the occurrences of \(\mathbf{*}\) 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 \(b\mathbf{v}\) 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 \(b_1\) and \(b_2\). If \(b_1\) holds, then \((s,t)\) can also be a member of the concretization. If \(b_2\) holds, then either of the two components of a pair of concrete states may be \(\varepsilon\) 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 \(\text{var}_{x,n}\), which are defined by means of Horn clauses. Formally, \(\tau \in d(x)\) at node \(n\) if \(\text{var}_{x,n}(\tau)\) holds. The values of \(b_1\) and \(b_2\) at node \(n\) are represented by means of the nullary predicates \(\text{public\_error}_n\) and \(\text{secret\_error}_n\), 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 \(n_0\). 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

\[\gamma(d)\]
tree automata here, but note that their languages can be defined by means of clauses of the form:

\[ p(\sigma_0). \text{ or } p(\sigma_2(X_1, X_2)) \leq p_1(X_1), p_2(X_2). \]

Above, \( p, p_1, \) and \( p_2 \) are unary predicates corresponding to the states of the automaton, and \( \sigma_0 \) and \( \sigma_2 \) are nullary and binary constructors respectively.

\[
\begin{align*}
\text{qPlist(patientRecord)} &= \text{id(id)} \\
\text{qPlist(patientRecord)} &= \text{qPlist(#)} \\
\text{qBV(value)} &= \text{Name(name)} \\
\text{qBV(value)} &= \text{Health(health)} \\
\text{qStar(*)} &= \text{qEmpty(#)}
\end{align*}
\]

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):

\[
\begin{align*}
\text{qPlist(patientRecord(L,R))} &\leq \text{id(L), qPlist(R).} \\
\text{qPlist(#)} &\leq \text{qBV(L), qName(R).} \\
\text{qName(name(L,R))} &\leq \text{qBV(L), qHealth(R).} \\
\text{qHealth(health(L,R))} &\leq \text{qStar(L), qEmpty(R).} \\
\text{qBV(value)} &\leq \text{qStar(*), qEmpty(#).}
\end{align*}
\]

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 id 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 * 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 \( n_0 \) and for this variable is defined by: \( \text{var}_{\text{pList}, n_0}(X) \leq \text{qPlist}(X) \).

In the following, we describe how the abstract state transformers \([f, g]_2\) 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 \( \Sigma_2 \) 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 \( [x := c_1(x_1, \ldots , x_k), y := c_2(y_1, \ldots , y_m)] \). 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:

\[
\text{var}_{z, n'}(X) \leq \text{var}_{z, n}(X).
\]

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

- For edges with label \( (x := #, x := #) \) we have \( \text{var}_{x, n'}(#) \leq \text{var}_{y, n}(\_). \) for all variables \( y \) occurring in the program, where ‘’ denotes an anonymous logical variable. The implication is required to ensure that \( # \) is added to the predicate \( \text{var}_{z, n'} \) only if \( n \) may be reachable. We consider a node \( n \) as unreachable, if there is no variable \( x \) and tree \( \tau \) so that \( \text{var}_{x, n}(\tau) \) holds.
- For edges with \( (x := y, x := y) \) we have: \( \text{var}_{x, n'}(X) \leq \text{var}_{y, n}(X). \)
- For edges with \( (x := \sigma_2(y, z), x := \sigma_2(y, z)) \) we have:

\[
\text{var}_{x, n'}(\sigma_2(L, R)) \leq \text{var}_{y, n}(L), \text{var}_{z, n}(R).
\]

- For edges with \( (x := y/1, x := y/1) \) we have \( \text{var}_{x, n'}(L) \leq \text{var}_{y, n}(\sigma_2(L, R)). \) 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 \( \text{id} := \text{pList}/1 \) we can assign the head of the list into variable \( \text{id} \). The implication defining the abstract value of variable \( \text{id} \) after the assignment is \( \text{var}_{\text{id}, n}(L) \leq \text{var}_{\text{pList}, n}(\text{patientRecord}(L, \_)). \) 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:

\[
\text{public_error}_n \leq \text{var}_{y, n}(\_).
\]

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

By edges with \( (f, f) \) where \( f = x := \lambda t(x_1, \ldots , x_k) \), it needs to be examined whether the arguments of the function \( \lambda t \) contain secret. The implications below are used for the purpose, where the second and third lines are defined for all \( \sigma_2 \in \Sigma_2 \):

\[
\begin{align*}
\text{secret}(\_). & \leq \text{var}_{\text{secret}, n}(\_). \\
\text{secret}(\sigma_2(L, \_)) &\leq \text{secret}(L). \\
\text{secret}(\sigma_2(\_ , \_)) &\leq \text{secret}(\_).
\end{align*}
\]

Concerning the resulting value of \( x \) we have:

\[
\begin{align*}
\text{var}_{x, n'}(bv) &\leq \text{var}_{x_1, n}(\_), \text{var}_{x_2, n}(\_), \\
&\ldots, \text{var}_{x_k, n}(\_). \\
\text{var}_{x, n'}(\_). &\leq \text{var}_{x_1, n}(X), \text{secret}(X), \\
&\text{var}_{x_2, n}(\_), \ldots, \\
&\text{var}_{x_k, n}(\_).
\end{align*}
\]

According to implication (8), the value of \( x \) at node \( n' \) will potentially be \( bv \) if all of the arguments \( x_1, \ldots , x_k \) of \( \lambda t \) 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 *. There is an implication of the form (9) defined for all arguments \( x_i \) of \( \lambda t \).
By edges of the form \((x:=e(x_1, \ldots, x_k), \text{skip})\) and \((\text{skip}, x:=e(x_1, \ldots, x_k))\) we have:

\[
\text{var}_{x,n}(\ast) \leftarrow \text{var}_{x_1,n}(\_), \ldots, \text{var}_{x_k,n}(\_).
\]

If the effect of an edge consists of an assignment and a \text{skip} command, then in the resulting abstract state the value of the variable on the left hand side becomes \ast 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

\[
\text{var}_{x,n}(\_):=\text{var}_{x,n}(\_), \ldots, \text{var}_{x_k,n}(\_).
\]

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

\[
\text{var}_{x,n'}(\delta(L,R)) \leftarrow \text{var}_{x,n}(\delta(L,R)).
\]

Additionally, we need to define \((10)\) and \((11)\) so that \(\delta(X,Y)\) is replaced by \ast, and if \(\sigma \neq \#\) then by \# too.

By edges labeled \((b,b)\), \((b,\text{skip})\) or \((\text{skip},b)\), where \(b = \lambda_0(x_1, x_2, \ldots, x_k)\), the values of all variables \(y\) occurring in the program are propagated the following way:

\[
\text{var}_{y,n'}(X) \leftarrow \text{var}_{y,n}(X), \text{var}_{x_1,n}(\_), \ldots, \text{var}_{x_k,n}(\_).
\]

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 \((\text{top}(x)\Rightarrow \sigma, \text{top}(x)\Rightarrow \sigma)\), then the following clauses are defined to propagate the values of variables \(y \neq x\) if \(\sigma \in \Sigma_2:\n
\[
\text{var}_{y,n}(X) \leftarrow \text{var}_{y,n}(X), \text{var}_{x,n}(\sigma(\_)), \ldots, \text{var}_{x_k,n}(\_). \tag{10}
\]

\[
\text{var}_{y,n}(X) \leftarrow \text{var}_{y,n}(X), \text{var}_{x,n}(\_)(\ast). \tag{11}
\]

The value of the variable \(x\) is propagated as well:

\[
\text{var}_{x,n}(\sigma(L,R)) \leftarrow \text{var}_{x,n}(\sigma(L,R)). \tag{11}
\]

\[
\text{var}_{x,n}(\ast) \leftarrow \text{var}_{x,n}(\ast). \tag{12}
\]

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

For edges with \((\text{top}(x)\Rightarrow \sigma, \text{skip})\) or \((\text{skip}, \text{top}(x)\Rightarrow \sigma)\) \((10)\) and \((11)\) need to be repeated, and in addition

\[
\text{var}_{x,n}(\ast) \leftarrow \text{var}_{x,n}(\ast) \tag{12}.
\]

By edges with \((-\text{top}(x)\Rightarrow \sigma, -\text{top}(x)\Rightarrow \sigma), (-\text{top}(x)\Rightarrow \sigma, \text{skip})\) or \((\text{skip}, -\text{top}(x)\Rightarrow \sigma)\), 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\}:

\[
\text{var}_{y,n'}(X) \leftarrow \text{var}_{y,n}(X), \text{var}_{x,n}(\delta(\_)), \ldots. \tag{13}
\]

In other words, it is checked whether the input variable \(y\) 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.

By edges labeled \((b,b)\), \((b,\text{skip})\) or \((\text{skip},b)\), where \(b = \lambda_0(x_1, x_2, \ldots, x_k)\), the values of all variables \(y\) occurring in the program are propagated the following way:

\[
\text{var}_{y,n'}(X) \leftarrow \text{var}_{y,n}(X), \text{var}_{x_1,n}(\_), \ldots, \text{var}_{x_k,n}(\_).
\]

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.

By edges labeled \((b,b)\), \((b,\text{skip})\) or \((\text{skip},b)\), where \(b = \lambda_0(x_1, x_2, \ldots, x_k)\), the values of all variables \(y\) occurring in the program are propagated the following way:

\[
\text{var}_{y,n'}(X) \leftarrow \text{var}_{y,n}(X), \text{var}_{x_1,n}(\_), \ldots, \text{var}_{x_k,n}(\_).
\]

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.

By edges labeled \((b,b)\), \((b,\text{skip})\) or \((\text{skip},b)\), where \(b = \lambda_0(x_1, x_2, \ldots, x_k)\), the values of all variables \(y\) occurring in the program are propagated the following way:

\[
\text{var}_{y,n'}(X) \leftarrow \text{var}_{y,n}(X), \text{var}_{x_1,n}(\_), \ldots, \text{var}_{x_k,n}(\_).
\]

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.
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.

```plaintext
subroutine query {
    found := 'ff'; // var found,(3,3) (bu).
    while found = 'ff' {
        id := pList/1; // var id,(5,5) (X) \( \equiv \) qId(X).
        idVal := id/1; // var idVal,(6,6) (X) \( \equiv \) qBV(X).
        if idVal = patientId {
            found := 'tt'; // var found,(8,8) (X) \( \equiv \) qBV(X).
            name := id/2; // var name,(9,9) (X) \( \equiv \) qName(X).
            nameVal := name/1;
            // var nameVal,(11,11) (X) \( \equiv \) qBV(X).
            pList := pList/2; // var pList,(3,3) (X) \( \equiv \) qPlist(X).
        } else {
            pListRev := patientRecord(id,pListRev);
            // var pListRev,(16,16) (X) \( \equiv \) qPlist(X).
            pList := pList/2;
            // var pList,(3,3) (X) \( \equiv \) qPlist(X).
        }
    }
} else {
    pListRev := patientRecord(id,pListRev);
    // var pListRev,(16,16) (X) \( \equiv \) qPlist(X).
    pList := pList/2;
    // var pList,(3,3) (X) \( \equiv \) qPlist(X).
}
}
```

Listing 3: A subroutine restore, as well as a modification to the query routine from Listing 2. The listing shows how the database can be restored if the patient identifier is known.

```plaintext
subroutine restore {
    id := idVal/1; // var idVal,(11,11) (X) \( \equiv \) qId(X).
    pList := pListRev/2; // var pListRev,(27,27) (X) \( \equiv \) qPlist(X).
}
```

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 (10). 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 \( \ast \), 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 `pp2cfg(p_1,p_2,n_1,n_2)` 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 `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 3. b) Joining Tables. This program implements the join of two tables as published in [16] and in [8]. 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-
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™ 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.

| 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

6. RELATED WORK

This paper relies on the observation [2, 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., [4, 11, 13, 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. [6] 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., [6, 27, 28, 78]. Another 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, 38] where the security labels 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 [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 [1,13,17,29]. 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 H1. 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 H1 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


