Runtime Enforcement of Information Flow Security in Tree Manipulating Processes

Máté Kovács and Helmut Seidl

Technische Universität München, Germany

Abstract. We consider the problem of enforcing information flow policies in XML manipulating programs such as Web services and business processes implemented in current workflow languages. We propose a runtime monitor that can enforce the secrecy of freely chosen subtrees of the data throughout the execution. The key idea is to apply a generalized *constant propagation* for computing the public effect of branching constructs whose conditions may depend on the secret. This allows for a better precision than runtime monitors which rely on tainting of variables or nodes alone. We demonstrate our approach for a minimalistic tree manipulating programming language and prove its correctness w.r.t. the concrete semantics of programs.

Keywords: semi-structured data, information flow control, runtime enforcement

1 Introduction

As the application of computer based workflow and information storage solutions becomes ubiquitous in today's practice, the successful operation of organizations depends heavily on the correct functionality of these systems. In particular, invaluable pieces of information are stored and manipulated by computer systems that are in the same time connected to the Internet, so it is our valid expectation that the security of our data should be guaranteed. Yet, there are numerous problems and challenges that need to be tackled before we can achieve this goal.

This paper is concerned with the problem of enforcing information flow security in contemporary business workflows implemented e.g. using the Web Services Business Process Execution Language (BPEL) [5]. One common property of these workflows is that the data they manipulate is inherently semi-structured. The common data representation and exchange format is XML document trees, especially if Web services are used for communication. In our opinion, the challenge in this environment is to provide sufficient flexibility to enable the specification of secrecy in terms of parts of document trees, besides enforcing information flow policies.

This work was partially supported by the German Research Foundation (DFG) under the project SpAGAT (grant no. FI 936/2-1) in the priority program "Reliably Secure Software Systems – RS3".

There are established solutions for providing information flow control in structured and object-oriented languages like the Java extension Jif [4, 23], and ValSoft/Joana [15]. Further results in this area include among others [8, 10, 13, 30]. These approaches define information flow policies in terms of variables. However, if a complex data structure consisting of pieces with different security levels is encoded into the value of a single variable, policies associating security levels with variables might be too restrictive, because they would be forced to consider the secrecy level of the complete data structure to be equivalent with that of the most confidential member of the complete structure. The solution we propose associates secrecy levels to specific positions in the tree structured data during runtime.

The authors of [28] note that runtime approaches [13, 30, 26] are on the rise again, because they can be more permissive than static solutions, while providing the same guarantees. In our case this statement especially holds, because our monitor takes advantage of the fact that during runtime data instances are available. In principle, our monitor executes programs in parallel to the operational semantics of the language, while maintaining a state which only depends on the public data. Accordingly, we call the state of the monitor the *public* or *low view*. The computation of the low view is challenging in the case, when the result of a branching construct, whose condition depends on the secret, is about to be computed. In this case we apply a dataflow analysis procedure, which is a refinement of constant propagation (see e.g. [29]) for handling semi-structured data. The key difference is the hierarchic nature of lattice elements, which aligns to our purpose of preventing information leakage in tree-manipulating programs. Moreover, we gain precision by only considering a modification of a subtree inside a secret-dependent branch as potentially secret, if it does not occur in the other alternative as well, and thus must be excluded from the public view. In summary, our paper provides the following innovations:

- A runtime monitor is introduced to support the specification of information flow policies in terms of tree-like data and their enforcement.
- The enforcement mechanism applies a generalized variant of constant propagation in order to compute the public view of the state at the end of branching instructions.

The paper is organized as follows. In Section 2 we introduce a minimalistic language for which the construction of the monitor is demonstrated. In Section 3 we illustrate the intuition behind our solution through an example, a fragment of a hypothetical paper submission system. We formalize the approach in Section 4, and in Section 5 we discuss the guarantees the monitor provides us. In Section 6 we relate our work to others and conclude.

2 Preliminaries

In order to minimize the formal overhead, we demonstrate our approach with a small programming language for manipulating trees. Here, we consider binary

trees only. This is no restriction in generality, since binary trees are in one-to-one correspondence with unranked trees. Unranked trees in turn can be considered as the natural internal representation of XML documents.

Definition 1 (Binary trees). *The set of binary trees* $\mathfrak{B}_{\Sigma,\{\#\}}$ *over the finite set of binary alphabet elements* Σ *and the set of nullary alphabet elements* $\{\#\}$ *is defined by the language:*

$$t ::= \# \mid \beta(t_1, t_2)$$

where $\beta \in \Sigma$ *and* $t_1, t_2 \in \mathfrak{B}_{\Sigma,\{\#\}}$.

The left hand side of Figure 1 displays an unranked document tree representing a scientific publication in a database, while the right hand side shows its binary equivalent in the *first-child/next-sibling* encoding. The binary tree $\beta(t_1, t_2)$ is interpreted as an unranked forest, where the root of the leftmost tree is labeled β. Its content is the unranked variant of t_1, while the forest on its right hand side is the unranked variant of t_2. The only nullary node labeled # represents the empty forest.

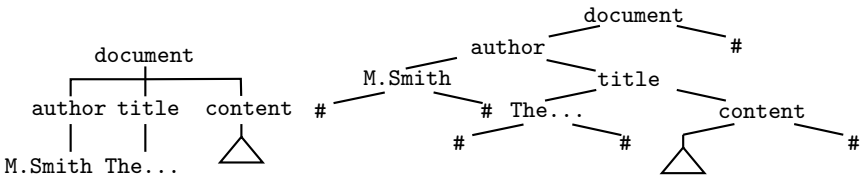

Fig. 1. Encoding unranked trees into binary trees

Our goal is to design a monitor which enforces confidentiality in programs manipulating tree structured data. In principle, the presented models may be applied also to interactive processes. In this paper, though, we disregard interactive aspects, like communication primitives. Instead, we focus on the key aspects of the monitor construction. Therefore, we assume that the input processed by programs is given in the initial configuration, and the result is presented in the final configuration.

(tree expressions)	$e ::= x \mid \texttt{\#} \mid \beta\texttt{(}x_1\texttt{,}x_2\texttt{)} \mid x\texttt{/1} \mid x\texttt{/2}$
(boolean expressions)	$b ::= \texttt{top(}x\texttt{)=}\alpha$
(program)	$p ::= \varepsilon \mid c\texttt{;}p$
(commands)	$c ::= x\texttt{<-}e \mid \texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ p \mid \texttt{while}\ b\ \texttt{do}\ p$

Fig. 2. A minimal language optimized for tree manipulation

A grammar for our minimalistic programming language is shown in Figure 2. A tree expression is the content of a variable x, the nullary node #, or a binary

tree composed of a new root labeled β having the contents of variables x_1 and x_2 as subtrees. The expressions x`/1` and x`/2` refer to the first and second subtree of the tree stored in variable x. Boolean expressions may test the label of the root of the tree stored in a variable. A program, generated by the nonterminal p, is a possibly empty sequence of commands. A command can be an assignment x`<-`e of the value of a tree expression e to the variable x, a conditional execution of alternative programs `if`, or an iteration `while`.

The semantics of the language is defined by transition relations $cfg \to_\rho cfg'$ between configurations of the form $\langle p, \sigma \rangle$, where p is the program to be executed on the state σ. In case of final configurations, where $p = \varepsilon$ we simply write σ instead of $\langle \varepsilon, \sigma \rangle$. The state $\sigma : (Var \to \mathfrak{B}_{\Sigma,\{\texttt{\#}\}}) \cup \{\lightning\}$ is a mapping from the set of variables of the program to binary trees, or the error state, denoted by $\lightning$, symbolizing that a runtime error has occurred during the execution.

The semantics of assignments and boolean expressions is shown in Figure 3. A boolean expression transforms a state into a boolean value (`t` or `f`), while an assignment transforms the state before the assignment into the state after the assignment. The new state σ' is equal to the original σ in all variables except for the one on the left hand side of the assignment. Such a modification of the mapping σ at argument x is denoted by $\sigma' = \sigma[x \mapsto v]$, where v is the new value. This new value for an assignment is obtained by evaluating the expression on the right hand side within the state σ.

$$[\![\texttt{top(}x\texttt{)=}\alpha]\!]\sigma = \begin{cases} \texttt{t} & \alpha \in \Sigma \text{ and } \sigma(x) = \alpha(t_1, t_2) \text{ for some } t_1 \text{ and } t_2 \text{ or} \\ & \alpha = \texttt{\#} \text{ and } \sigma(x) = \texttt{\#} \\ \texttt{f} & \text{otherwise} \end{cases}$$

$$[\![x\texttt{<-}y]\!]\sigma = \sigma[x \mapsto \sigma(y)] \qquad [\![x\texttt{<-\#}]\!]\sigma = \sigma[x \mapsto \texttt{\#}]$$

$$[\![x\texttt{<-}\beta\texttt{(}x_1\texttt{,}x_2\texttt{)}]\!]\sigma = \sigma[x \mapsto \beta(\sigma(x_1), \sigma(x_2))]$$

$$[\![x\texttt{<-}y\texttt{/1}]\!]\sigma = \begin{cases} \sigma[x \mapsto t_1] & \text{if } \sigma(y) = \beta(t_1, t_2) \text{ for some label } \beta \\ \lightning & \text{otherwise} \end{cases}$$

$$[\![x\texttt{<-}y\texttt{/2}]\!]\sigma = \begin{cases} \sigma[x \mapsto t_2] & \text{if } \sigma(y) = \beta(t_1, t_2) \text{ for some label } \beta \\ \lightning & \text{otherwise} \end{cases}$$

Fig. 3. The semantics of assignments and boolean expressions

When dealing with structured data, runtime errors cannot be excluded. When processing binary trees, an error occurs, if a subtree of a leaf is about to be accessed by expressions x`/1` or x`/2`. In this case the new state is the error state ($\lightning$). The error state is atomic, it does not map variables to values any more. We assume on the other hand, that each variable has already received a value in the initial configuration. Accordingly, no error can be caused by a variable access during the execution.

The semantics of the programming language is shown in Figure 4. In the condition parts of the rules we use the relation $\to_\rho^*$, which denotes the reflexive

and transitive closure of $\rightarrow_\rho$. A central rule of the semantics is S, which is responsible for executing a program, in other words a sequence of commands. The remaining rules define the effects of individual commands. By rule E the error state is not modified by any command. Instead, it is passed over to the next command in the sequence, or to the final configuration. In case the state is not erroneous, the execution of an assignment is specified by the rule A. The rules WT and WF execute iterations, IT and IF execute conditional selections of alternative programs as it is usual in other structured programming languages.

$$\text{E: } \frac{\sigma = \lightning}{\langle c, \sigma\rangle \rightarrow_\rho \lightning} \quad \text{A: } \frac{\sigma \neq \lightning}{\langle x\texttt{<-}e, \sigma\rangle \rightarrow_\rho [\![x\texttt{<-}e]\!]\sigma} \quad \text{S: } \frac{\langle c, \sigma\rangle \rightarrow_\rho^* \sigma'}{\langle c\texttt{;}p, \sigma\rangle \rightarrow_\rho \langle p, \sigma'\rangle}$$

$$\text{WT: } \frac{\sigma \neq \lightning \quad [\![b]\!]\sigma = \texttt{t} \quad \langle p_t, \sigma\rangle \rightarrow_\rho^* \sigma'}{\langle \texttt{while}\ b\ \texttt{do}\ p_t, \sigma\rangle \rightarrow_\rho \langle \texttt{while}\ b\ \texttt{do}\ p_t, \sigma'\rangle}$$

$$\text{WF: } \frac{\sigma \neq \lightning \quad [\![b]\!]\sigma = \texttt{f}}{\langle \texttt{while}\ b\ \texttt{do}\ p_t, \sigma\rangle \rightarrow_\rho \sigma}$$

$$\text{IT: } \frac{\sigma \neq \lightning \quad [\![b]\!]\sigma = \texttt{t} \quad \langle p_t, \sigma\rangle \rightarrow_\rho^* \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ p_t\ \texttt{else}\ p_f, \sigma\rangle \rightarrow_\rho \sigma'} \quad \text{IF: } \frac{\sigma \neq \lightning \quad [\![b]\!]\sigma = \texttt{f} \quad \langle p_f, \sigma\rangle \rightarrow_\rho^* \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ p_t\ \texttt{else}\ p_f, \sigma\rangle \rightarrow_\rho \sigma'}$$

Fig. 4. The semantics of the programming language

3 The Runtime Monitor

Since the seminal paper of Denning [11], the secrecy level of data is usually captured in terms of lattices. The simplest form of this lattice is $L \sqsubseteq H$, which allows to specify that pieces of information belonging to the lattice element L should not depend on those belonging to H.

Similarly to other runtime monitors e.g. [30, 13, 28, 26], in order to enforce information flow properties, we extend the configuration of the semantics of the language with an additional member, which maintains the secrecy information of the data stored in the state. This new member $D : (Var \rightarrow \mathfrak{B}_{\Sigma,\{\#,\square\}}) \cup \{\bot, \top, \lightning\}$, referred to as *monitor state*, assigns to every variable either a binary tree having the extra nullary alphabet element $\square$, or is one of the symbols $\bot$, $\top$ and $\lightning$. Intuitively, $D(x)$ stores the *public* upper part i.e., an upper part of the current value of x, which belongs to the security lattice element L, where the symbol $\square$ indicates subtrees possibly depending on the secret, and thus belonging to H. For this reason, we also call D the *public view*, because it only contains definitely public information. The monitor recalculates this value in parallel to the semantics for each configuration, and the result of the computation for principals belonging to the security lattice element L is presented by the final public view.

In the next paragraphs, we informally illustrate the functionality of the runtime monitor by an example. The code fragment in Listing 1 could be part of a paper submission system distributing the papers to reviewers. Let us suppose that reviewer 2 declared a conflict of interest with the author A. Mustermann, and therefore the distribution system is not allowed to send information to him about the content. Therefore, from the point of view of reviewer 2, the information on the topic of the paper of A. Mustermann is secret too. Let us suppose that the runtime monitor reaches line 3 of Listing 1 with monitor state:

$$D_0 = \{..., \texttt{topic} \mapsto \square, \texttt{rev1} \mapsto \#, \texttt{rev2} \mapsto \texttt{listElem}(\texttt{document}(...), \#), ...\}$$

```
1  empty<-#;
2  if top(author)=A_Mustermann then
3  if top(topic)=Databases then
4         rev2<-conflict(empty,rev2); rev1<-listElem(doc,rev1);
5  else   rev2<-conflict(empty,rev2); rev3<-listElem(doc,rev1);
6  ; else ;
```

Listing 1. Branching on a secret value

Because the conditional expression depends on the secret, constant propagation is carried out on this branching command. We can identify the value $\square$ with the top element of constant propagation expressing that the value is not constant and therefore, may leak information about the secret. After executing the branches we get:

$$\begin{aligned} D_{\texttt{then}} = \{..., \texttt{rev1} \mapsto \texttt{listElem}(\texttt{document}(...), \#), \\ \texttt{rev2} \mapsto \texttt{conflict}(\#, \texttt{listElem}(\texttt{document}(...), \#)), ...\} \\ D_{\texttt{else}} = \{..., \texttt{rev1} \mapsto \#, \texttt{rev2} \mapsto \texttt{conflict}(\#, \texttt{listElem}(\texttt{document}(...), \#)), ...\} \end{aligned}$$

After the join computation we have:

$$D = \{..., \texttt{rev1} \mapsto \square, \texttt{rev2} \mapsto \texttt{conflict}(\#, \texttt{listElem}(\texttt{document}(...), \#)), ...\}$$

Computing the join of two states can be done by replacing the values of variables where the two states differ, with the symbol $\square$. In this way, it is guaranteed that the monitor state after the branching construct is independent of the secret.

For the join computation, however, it is not necessary to replace the entire value of a variable with $\square$ if the two values differ only for certain subtrees. Figure 5 illustrates the join computation for the values of variable `authors` in monitor states $D_{\texttt{then}}$ and $D_{\texttt{else}}$ in such a situation. The variable contains a list of authors and their documents that they submitted. Let us suppose that the order of two authors has been accidentally exchanged in one of the two secret-dependent conditional branches. By computing the join, we only need to replace those members of the list which were exchanged, but we can leave the others as

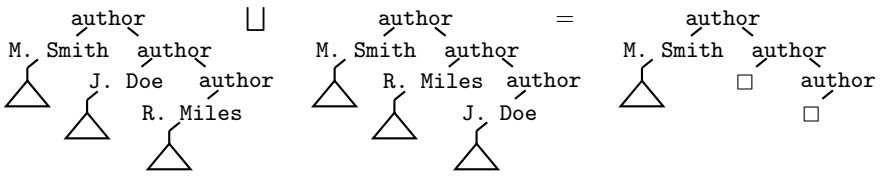

Fig. 5. The join on document trees, where the leaves labeled # are omitted for the sake of simplicity.

they are. In this way, we take the semi-structured nature of data into account and gain additional precision.

Another advantage of our approach is the following. Because in the code fragment of Listing 1 variable `rev2` is assigned in a secret branch, many information flow analyzers would consider its value secret. The solutions similar to the type system of Volpano et al. [31] like Jif [23, 4], SIF [10] and Paralocks [8] do so because the variable `rev2` has been assigned in an environment, where the program counter depends on the secret and therefore is high. Similarly behave runtime monitors [30, 13, 28, 26] for the same reason. The program slicing [17] based solutions like Joana [15] do so, because of the control dependency edges from the conditional expressions to assignments. Our idea is based on the observation that in the final configuration the value of the variable `rev2` is independent of the value of `topic`. This could happen, perhaps, because the program noticed by the embedding branching decision, that the content of the paper is secret and behaved correctly. Accordingly, the observation of `rev2` does not give us information on the secret value. Our runtime monitor would consider the value of `rev2` as public, because it determines the confidential parts of values by means of the join computation after exiting from branching commands depending on secret values. There are approaches based on bisimulation, e.g. [21, 18], allowing public assignments in secret branches, if the equivalence of the public effects of these branches is proved. Because program equivalence is in general undecidable, these solutions rely on syntactic approximations. In our solution if programs P and Q are equivalent, they do not read confidential variables, and they terminate, then the result of `if secret then` P `else` Q`;` is recognized public regardless of the syntactic representation of P and Q.

In the next section we formally elaborate the ideas introduced here.

4 Formal Treatment of the Monitor

In order to describe the runtime monitor formally, we need some more definitions. In the following, we view a tree t as a mapping from its positions $Pos(t)$ to the alphabet Σ of binary symbols or # and □, where the domain is a prefix closed subset of $\{1, 2\}^*$ with the additional property that if $p_i 2 \in Pos(t)$ then $p_i 1 \in Pos(t)$ too. Accordingly, we use the notation $t(p)$ to refer to the alphabet element at position p of the tree t. If a node p of t has no successors, then $t(p)$

either equals # or □. We denote the *subtrees* rooted at the first and the second child of the root of t with $t/1$ and $t/2$ respectively.

Definition 2 (Preorder of trees). *If $t_1, t_2 \in \mathfrak{B}_{\Sigma,\{\#,\square\}}$ then $t_1 \sqsubseteq t_2$ holds if one of the following is true:*

- $t_2(\varepsilon) = \square$.
- $t_2(\varepsilon) \neq \square$ *and* $t_1(\varepsilon) = t_2(\varepsilon)$, *furthermore, if* $t_1(\varepsilon) \neq \texttt{\#}$ *then* $t_1/1 \sqsubseteq t_2/1$ *and* $t_1/2 \sqsubseteq t_2/2$.

In Definition 2 the symbol □ occurs as an additional nullary element, which represents a secret subtree in the public view. Similarly to the state, the monitor state can also be erroneous, denoted by $\lightning$, meaning that the execution reached an inconsistent situation. It is also possible that the error state itself depends on the secret. This happens for instance, if one conditional branch of a decision depending on the secret exhibits an error, while the other does not. We have introduced the top element $\top$ to represent this case. For the monitor state, a dataflow analysis will be performed to approximate the public view after a secret-dependent branching construct. For this analysis, a bottom element (denoting unreachability) comes in handy to obtain a complete lattice (see Definition 3).

Definition 3 (Complete lattice of monitor states). *The complete lattice of monitor states is $\mathbb{D} = (Var \to \mathfrak{B}_{\Sigma,\{\#,\square\}}) \cup \{\lightning, \top, \bot\}$. For any $D_1, D_2 \in \mathbb{D}$ the relation $D_1 \sqsubseteq D_2$ holds if one of the following is true:*

- $D_1 = \bot$
- $D_2 = \top$
- $D_1 = \lightning$ *and* $D_2 = \lightning$
- *If* $D_1, D_2 \notin \{\lightning, \top, \bot\}$ *then for all variables x it holds that* $D_1(x) \sqsubseteq D_2(x)$ *according to Definition 2.*

The idea of the monitored execution is to execute the state transformations on the real state and the monitor state in parallel. In order to do so we need to specify the semantics of expressions on the monitor states. Considering tree expressions, we obtain the monitor semantics if we exchange σ with D in Figure 3 with the extension that $[\![x\texttt{/1}]\!]D = [\![x\texttt{/2}]\!]D = \top$ if $D(x) = \square$. This is inevitable for the following reason. In place of □ in the monitor state, the real state may have any binary tree and thus, in particular, may equal to `#`. Therefore, depending on the secret, the result of the expression on the real state may be $\lightning$ or not. Therefore, the monitor state must be switched to $\top$.

As it is shown in Figure 6, a boolean expression is a transformation on the monitor state just as tree expressions are. Basically, if the boolean expression is true on the monitor state then it is the identity function, otherwise the result is $\bot$. The only exception is in the positive case ((1) and (2)) when the content of the variable x is transformed to the greatest tree not being □, for which the boolean expression holds.

The semantics of the monitored execution is defined in the form of relations $cfg \to_\gamma cfg'$ between configurations of the form $\langle p, n, (\sigma, D)\rangle$ where p is a program

$$[\![\texttt{top}(x)\texttt{=}\alpha]\!]D = \begin{cases} D & \text{if } D(x)(\varepsilon) = \alpha \\ D[x \mapsto \alpha(\square,\square)] & \text{if } D(x) = \square \text{ and } \alpha \neq \texttt{\#} \quad (1) \\ D[x \mapsto \texttt{\#}] & \text{if } D(x) = \square \text{ and } \alpha = \texttt{\#} \quad (2) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\neg\texttt{top}(x)\texttt{=}\alpha]\!]D = \begin{cases} D & \text{if } D(x)(\varepsilon) \neq \alpha \text{ or } D(x) = \square \\ \bot & \text{otherwise} \end{cases}$$

Fig. 6. Semantics of boolean expressions on the monitor state

to be executed and σ is the state of the execution. The member D is the public view of the state σ, which we call the monitor state, σ is called the real state. The member n is a natural number influencing the precision of the monitor when computing the public effect of branching constructs. Larger values correspond to enhanced precision and longer computation time. In the initial configuration $\langle p, n, (\sigma, D)\rangle$ it holds that $\sigma \sqsubseteq D$ and in case σ is not the error state, there are no nodes labeled $\square$ in the contents of its variables. The intuitive meaning of the relation $\sqsubseteq$ between the real state and the monitor state is that they agree on public values, and this is the property our monitor guarantees along the run.

As long as the monitored semantics does not execute a branching construct whose boolean expression depends on the secret, the monitored execution succeeds similarly to the original execution semantics shown in Figure 4. Assignments are executed in parallel on σ and on D according to the semantics in Figure 3 with the extension that $[\![x\texttt{/1}]\!]D = [\![x\texttt{/2}]\!]D = \top$ if $D(x) = \square$. The truth values of boolean expressions are determined based on the monitor state. If $[\![b]\!]D = \bot$, then we consider $\neg b$ to be true. In case $[\![b]\!]D \neq \bot$ and $[\![\neg b]\!]D \neq \bot$ simultaneously, we execute a branching construct, whose condition depends on the secret. In this case the result of the branching command on the real state is computed using the original semantics of Figure 4, the resulting monitor state is computed using a generalized constant propagation algorithm. This is the point, where the parameter n is used. Assume that the command c is a branching construct whose condition depends on the secret in configuration $\langle c\texttt{;}p, n, (\sigma, D)\rangle$. In this case we apply the generalized constant propagation on the command $c(n)$, which we construct based on c by replacing all occurrences of the command `while` b `do` p in the program text of c by `while(`n`)` b `do` p.

The *generalized constant propagation* algorithm is defined in Figure 7, which is basically the rule-based formalization of a syntax directed fixed point computation algorithm on the program text as it is presented in [7]. The lattice is the set of possible monitor states according to Definition 3.

The rules defining the functionality of assignment (MA), sequential execution of commands (MS), and the propagation of the states $\top$, $\bot$ and $\lightning$ (MCE) are very similar to rules A, S and E of the original semantics. The only difference is at rule MCE, which propagates the states $\bot$ and $\top$ unmodified as well.

The rule MI is responsible for computing the monitor state transformation belonging to an `if` command. It evaluates both branches with initial states $[\![b]\!]D$ and $[\![\neg b]\!]D$ and then joins the results.

$$\text{MCE: } \frac{D \in \{\top, \bot, \lightning\}}{\langle c, D\rangle \to_\mu D} \quad \text{MA: } \frac{D \notin \{\top, \bot, \lightning\}}{\langle x\texttt{<-}e, D\rangle \to_\mu [\![x\texttt{<-}e]\!]D} \quad \text{MS: } \frac{\langle c, D\rangle \to^*_\mu D'}{\langle c\texttt{;}p, D\rangle \to_\mu \langle p, D'\rangle}$$

$$\text{MI: } \frac{\begin{array}{c}\langle p_t, [\![b]\!]D\rangle \to^*_\mu D_1 \\ \langle p_f, [\![\neg b]\!]D\rangle \to^*_\mu D_2\end{array} \quad D \notin \{\top, \bot, \lightning\} \quad D' = D_1 \sqcup D_2}{\langle \texttt{if } b \texttt{ then } p_t \texttt{ else } p_f, D\rangle \to_\mu D'} \quad \text{MWF: } \frac{\begin{array}{c}D \notin \{\top, \bot, \lightning\} \\ [\![b]\!]D = \bot\end{array}}{\langle \texttt{while(}n\texttt{) } b \texttt{ do } p_t, D\rangle \to_\mu D}$$

$$\text{MWT: } \frac{D \notin \{\top, \bot, \lightning\} \quad [\![\neg b]\!]D = \bot \quad n > 0 \quad \langle p_t, [\![b]\!]D\rangle \to^*_\mu D'}{\langle \texttt{while(}n\texttt{) } b \texttt{ do } p_t, D\rangle \to_\mu \langle \texttt{while(}n-1\texttt{) } b \texttt{ do } p_t, D'\rangle}$$

$$\text{MWH: } \frac{\begin{array}{c}([\![\neg b]\!]D \neq \bot \wedge [\![b]\!]D \neq \bot) \vee n \leq 0 \\ D \notin \{\top, \bot, \lightning\} \quad \langle p_t, [\![b]\!]D\rangle \to^*_\mu D_1 \quad D' = D_1 \sqcup D \quad D' \not\sqsubseteq D\end{array}}{\langle \texttt{while(}n\texttt{) } b \texttt{ do } p_t, D\rangle \to_\mu \langle \texttt{while(}n-1\texttt{) } b \texttt{ do } p_t, D'\rangle}$$

$$\text{MWX: } \frac{\begin{array}{c}([\![b]\!]D \neq \bot \wedge [\![b]\!]D \neq \bot) \vee n \leq 0 \\ D \notin \{\top, \bot, \lightning\} \quad \langle p_t, [\![b]\!]D\rangle \to^*_\mu D_1 \quad D' = D_1 \sqcup D \quad D' \sqsubseteq D\end{array}}{\langle \texttt{while(}n\texttt{) } b \texttt{ do } p_t, D\rangle \to_\mu D}$$

Fig. 7. Generalized constant propagation

The rules MWT, MWF, MWH and MWX are used to compute the public effect of iterations. If the parameter n is zero, or the condition is secret-dependent, a fixed point is computed by rules MWH and MWX. If, however, the condition is independent of the secret, and n is greater then zero, the monitor executes the body of the loop iteratively by applying rules MWT and MWF. In the same time this might not terminate. So the purpose of n is to allow the user to specify how many times the monitor should apply the rule MWT before switching to the fixed point computation. In particular, setting the parameter n to zero in the initial configuration of the monitored execution $\langle p, n, (\sigma, D)\rangle$ amounts to choosing to omit the application of rules MWT and MWF, and use only the fixed point computation offered by rules MWH and MWX. In the same time this might result in an unnecessarily inaccurate monitor state.

Because the complete lattice of Definition 3 has the ascending chain condition, as Theorem 1 states below, the fixed point computation always terminates.

Theorem 1. *If there is σ' so that $\langle p, \sigma\rangle \to^*_\rho \sigma'$ and $\sigma \sqsubseteq D$, then there is a D' so that $\langle p, n, (\sigma, D)\rangle \to^*_\gamma (\sigma', D')$.*

Proof. The idea behind the proof is the following: If there is no branching construct executed having condition depending on the secret along the monitored execution, then the state transitions are carried out simultaneously on the real state and on the monitor state. In the case of a branching construct having secret-dependent condition, the public effect is computed by the algorithm in Figure 7. The only rule, which could be applied an unbounded number of times

is MWH, but because the lattice of Definition 3 has the ascending chain condition, the fixed point computation terminates. For the detailed proof please refer to [19].

5 Guarantees

In this section we formally discuss the guarantees provided by the runtime monitor.

Similarly to other language-based information flow controlling solutions [12, 23, 4, 27, 26, 15, 13, 30], our approach enforces a variant of Termination Insensitive Noninterference [6] tailored for our computational model. Accordingly, we do not consider covert channels like the timing channel, the heat channel, or the memory consumption channel, or any other channel that could result from the properties of a specific implementation or runtime environment.

Definition 4. *Program p satisfies Termination Insensitive Noninterference relative to the initial and final public views D and D' if and only if for all $\sigma_1, \sigma_2 \sqsubseteq D$ it is true that if*

- $\langle p, \sigma_1 \rangle \rightarrow_\rho^* \sigma_1'$ *and*
- $\langle p, \sigma_2 \rangle \rightarrow_\rho^* \sigma_2'$

then $\sigma_1' \sqsubseteq D'$ and $\sigma_2' \sqsubseteq D'$. In this case we say that D' is an appropriate final public view belonging to the program p and the initial public view D.

The monitored execution $\langle p, n, (\sigma, D) \rangle \rightarrow_\gamma^* (\sigma', D')$ computes a pair (σ', D') based on (σ, D), where D' is an appropriate final public view belonging to p and D. We may consider the public view D as an indistinguishability relation between any initial states σ^*, for which it holds that $\sigma^* \sqsubseteq D$. The meaning of the resulting monitor state D' is that by observing it, we do not gain information on which σ^* was the initial state. Accordingly, we can communicate parts or the entire final public view D' to principals having low security clearance (L), and we can consider nodes labeled $\Box$ as a default value that secret pieces of information have been replaced with. If $D' = \top$ then the observers of the public do not gain more information than the fact that the execution of the program terminated. Still, principals with high security clearance (H) may observe the resulting real state σ', and use the computed values.

The following theorem assures us that our monitor indeed computes an appropriate final public view belonging to the program and the initial public view:

Theorem 2. *If there are two initial states σ_1 and σ_2 so that $\sigma_1, \sigma_2 \sqsubseteq D$, then if*

- $\langle p, n, (\sigma_1, D) \rangle \rightarrow_\gamma^* (\sigma_1', D_1')$ *and*
- $\langle p, n, (\sigma_2, D) \rangle \rightarrow_\gamma^* (\sigma_2', D_2')$

then $D_1' = D_2' = D'$ and $\sigma_1' \sqsubseteq D'$ and $\sigma_2' \sqsubseteq D'$.

Proof. For the proof please refer to [19].

6 Related Work and Conclusion

This paper is related to two research areas. One is language based information flow security as we have discussed in Section 3, the other is formalization and verification of Web service compositions and business workflows. Much effort has been invested to find adequate formalisms to describe the functionality of service orchestrations and choreography, in particular, the BPEL [5] language, in order to enable formal rigor for development and to allow verification. The majority of the publications in this topic can be sorted into two groups. One [16, 2, 1, 24] applies formalisms based on Petri-nets to model workflows, the other [14, 20, 25, 32, 9] prefers algebraic calculi like the Π-calculus [22] as the basis for investigations. The authors of [3] and [33] present security related results using Petri-net based formalisms. A common property of these approaches is that they mostly focus on the control flow of orchestrations, sometimes with emphasis on error handling, whereas data values undergo severe abstractions. Data are either considered as atomic values, or completely disregarded by handling branching decisions as nondeterminism.

At the same time practical business processes mostly use XML documents for data representation, which are naturally semi-structured and unbounded in size. In the case of these processes, security is especially important, because of their distributed nature. For this reason we showed for a minimalistic, yet Turing-complete language, how to provide security guarantees without carrying out major abstractions on the data.

As a result we have presented an approach to enforce information flow control in tree manipulating processes. Since practical information flow policies refer to the structure of data, approaches where security specifications are bound to variables such as [4, 23, 15, 8, 10, 13, 30] may not suffice. In their paper [26], Russo et al. aim at a similar goal like us. In their formal model of JavaScript, they consider one DOM tree representing the data in a Web browser. However, their formalism is still quite different to ours. Their computational model operates on a single unranked tree using a pointer on one specific working node, and supports operations like insertion, modification and removal. Since their monitor maintains the security levels of the positions of the DOM tree during the run, it can be considered as a generalization of the idea of binding secrecy levels to variables towards trees. Our monitor, on the other hand, maintains the concrete values of public nodes making it possible to take the semantics of branches into account where the conditional depends on the secret and compute the public effect by means of a value-based comparison of the resulting states.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems 30(4), 245 – 275 (2005)
2. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press (2002)

3. Accorsi, R., Wonnemann, C.: Static information flow analysis of workflow models. In: Abramowicz, W., Alt, R., Fähnrich, K.P., Franczyk, B., Maciaszek, L.A. (eds.) ISSS/BPSC. LNI, vol. 177, pp. 194–205. GI (2010)
4. A.C.Myers, L.Zheng, S.Zdancewic, S.Chong, N.Nystrom: Jif:java + information flow (July 2001–2011), Software release. Located at `http://www.cs.cornell.edu/jif`
5. Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, C.K., Khalaf, R., Koenig, D., Marin, M., Mehta, V., Thatte, S., Rijn, D., Yendluri, P., Yiu, A.: 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)
6. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: ESORICS. Lecture Notes in Computer Science, vol. 5283, pp. 333–348. Springer (2008)
7. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. CoRR abs/cs/0701193 (2007)
8. Broberg, N., Sands, D.: Paralocks – role-based information flow control and beyond. In: POPL'10, Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (2010)
9. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Palsberg, J., Abadi, M. (eds.) POPL. pp. 209–220. ACM (2005)
10. Chong, S., Vikram, K., Myers, A.C.: SIF: enforcing confidentiality and integrity in web applications. In: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium. pp. 1:1–1:16. USENIX Association, Berkeley, CA, USA (2007)
11. Denning, D.E.: A lattice model of secure information flow. Commun. ACM 19(5), 236–243 (1976)
12. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM 20(7), 504–513 (1977)
13. Guernic, G.L.: Automaton-based confidentiality monitoring of concurrent programs. In: CSF. pp. 218–232. IEEE Computer Society (2007)
14. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: Sock: A calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC. Lecture Notes in Computer Science, vol. 4294, pp. 327–338. Springer (2006)
15. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security 8(6), 399–422 (Dec 2009)
16. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri nets. In: Aalst, W.M.P.v.d., Benatallah, B., Casati, F., Curbera, F. (eds.) Proceedings of the Third International Conference on Business Process Management (BPM 2005). Lecture Notes in Computer Science, vol. 3649, pp. 220–235. Springer-Verlag, Nancy, France (Sep 2005)
17. Horwitz, S., Prins, J., Reps, T.: On the adequacy of program dependence graphs for representing programs. In: POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 146–157. ACM, New York, NY, USA (1988)
18. Köpf, B., Mantel, H.: Transformational typing and unification for automatically correcting insecure programs. Int. J. Inf. Sec. 6(2-3), 107–131 (2007)

19. Kovács, M., Seidl, H.: Runtime enforcement of information flow security in tree manipulating processes (proofs). Tech. rep., Technische Universität München, Institut für Informatik (2011)
20. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for WS-BPEL. J. Log. Algebr. Program. 70(1), 96–118 (2007)
21. Mantel, H., Sands, D.: Controlled declassification based on intransitive noninterference. In: Chin, W.N. (ed.) APLAS. Lecture Notes in Computer Science, vol. 3302, pp. 129–145. Springer (2004)
22. Milner, R.: Communicating and Mobile Systems: the Π-calculus. Cambridge University Press (1999)
23. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: Proc. 26th ACM Symp. on Principles of Programming Languages (POPL). pp. 228–241 (1999)
24. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: WofBPEL: A tool for automated analysis of BPEL processes. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC. Lecture Notes in Computer Science, vol. 3826, pp. 484–489. Springer (2005)
25. Rouached, M., Godart, C.: Requirements-driven verification of WSBPEL processes. Web Services, 2007. ICWS 2007. IEEE International Conference on pp. 354–363 (July 2007)
26. Russo, A., Sabelfeld, A., Chudnov, A.: Tracking information flow in dynamic tree structures. In: Proceedings of the European Symposium on Research in Computer Security (ESORICS). LNCS, Springer-Verlag (September 2009)
27. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (January 14 2003)
28. Sabelfeld, A., Russo, A.: From dynamic to static and back: Riding the roller coaster of information-flow control research. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 09. Lecture Notes in Computer Science, vol. 5947, pp. 352–365. Springer (2009)
29. Seidl, H., Wilhelm, R., Hack, S.: Compiler Design: Analysis and Transformation. Springer (September, 2011)
30. Venkatakrishnan, V.N., Xu, W., DuVarney, D.C., Sekar, R.: Provably correct runtime enforcement of non-interference properties. In: Ning, P., Qing, S., Li, N. (eds.) ICICS. Lecture Notes in Computer Science, vol. 4307, pp. 332–351. Springer (2006)
31. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security 4(2/3), 167–188 (1996)
32. Wirsing, M., Clark, A., Gilmore, S., Hölzl, M., Knapp, A., Koch, N., Schroeder, A.: Semantic-based development of service-oriented systems. In: et al., E.N. (ed.) Proc. 26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems(FORTE'06), Paris, France. pp. 24–45. LNCS 4229, Springer-Verlag (2006)
33. Wolter, C., Miseldine, P., Meinel, C.: Verification of business process entailment constraints using SPIN. In: Massacci, F., Jr., S.T.R., Zannone, N. (eds.) ESSoS. Lecture Notes in Computer Science, vol. 5429, pp. 1–15. Springer (2009)