An end-to-end security model for Adaptive Service-oriented applications

Takoua Abdellatif, Marius Bozga

An end-to-end security model for Adaptive Service-oriented applications

Takoua Abdellatif [1] and Marius Bozga[2]

[1] Univ. of Carthage, Tunisia Polytechnic School, SERCOM. 2078, La Marsa, Tunisia
Takoua.Abdellatif@ept.rnu.tn

[2] CNRS / Univ. of Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
Marius.Bozga@imag.fr

Abstract. In this paper, we present E2SM, an End-to-End Security Model and a set of algorithms to protect data confidentiality in complex adaptive Service-oriented applications SOA. Starting from initial and intuitive business security constraints' settings, E2SM synthesizes a complete security configuration that is formally verified. E2SM is adapted to dynamic security constraints' modifications and to services' architecture reconfiguration. Thanks to its compositional verification, only impacted services' security is rechecked which makes E2SM suitable to adaptive and scalable SOA.

Keywords: Configuration synthesis, Security Formal checking, SOA, adaptation.

1 Introduction

SOA applications are generally built from existing services and based on standard tools for service composition like Web Services (WS) [1]. These applications are deployed in various domains like e-health systems, e-commerce and social networks. They exchange critical information between mistrusted parties and networks and have to preserve people's privacy and business data confidentiality. Several standards and tools [2][3][4] are currently used to secure WS communications. They are generally based on access control policies that focus on point-to-point communications. Nevertheless, end-to-end security is required and implies tracking information flow through all the system services and analyzing data dependencies. Typically, forwarding confidential data to unauthorized services has to be detected. Consequently, data dependence has to be tracked and explicitly exposed; otherwise, security composition can induce security leaks called interference [5].

An application free of this problem is said to satisfy the **non-interference** property [8]. To check that a WS satisfies non-interference, a classical technique consists in classifying WS data with respect to its security level and verifying that data with low-level security constraint (like public data) is not influenced or calculated from any data with high-level security constraint (like confidential data). Furthermore, the secu-

rity classification model has to consider authorization rules and trust between services and consider the network not unique as potential attacker. Indeed, current SOA applications are generally composed of loosely coupled services that come from different providers. Tracking data flow to non-authorized services is not an easy task in a distributed system [6].

Standard orchestration languages like BPEL (Business Process Execution Language) [7] are practical to build complex composed WS and to track their execution for reliability reasons mainly. Nevertheless, these languages, as well as WS security standards, provide no way to define data security classification and to check non-interference property. There is a clear need for a new security model and a **practical** tool that helps administrators setting intuitive security configuration parameters and synthesizes automatically the whole security configuration. The tool has to be **robust** proving formally that the security configuration satisfies security policies. It has to be **scalable**, that is when the number of services' activities is huge or when the number of composed services is important, security configuration is calculated within an acceptable reasonable time. Furthermore, security configuration has to be **adaptable** to special cases like emergency where security constraints have to be relaxed. For example, in a home gateway application, if a fire is detected, the information has to reach a remote monitor service in a short time even though such information is not protected with cryptography encryption. More generally, a compromise between security constraints and functionality needs has to be found and relaxing some information secrecy is sometimes needed to reach this compromise. This feature is called declassification and is still a hard problem in security research field and is rarely implemented in real applications like SOA [9] [10]. Another aspect of adaptation is when the system architecture is modified. Typically, removing or adding new services or modifying some security configuration settings may require the security configuration synthesis that has to be executed rapidly not to slow down the adaptation time.

In this paper, we propose E2SM (End-to-End Security Model), a security model for composed and adaptive SOA applications construct as a composition of BPEL processes. The main idea of E2SM is to abstract BPEL processes to a set of graphs to show data dependences and starting from a few initial configuration settings, the whole system configuration is generated while checking its non-interference property. Non-interference checking is modular: to check that a hierarchical BPEL process is secure, it is sufficient to check that each involved BPEL process is secure. Furthermore, E2SM assists users (service designers, developers or administrators) to set-up business oriented security constraints at a high level, checks constraints coherence and synthesizes a complete secure configuration. If the user's security constraints induce non-secure configuration, the user is guided to correct his initial configuration by relaxing some constraints or modifying his initial configuration. Dynamically, E2SM checks any security re-configuration. It tracks the affected service parts and regenerates the new security configuration in an acceptable time. Our solution is practical since users do not need to be security experts, robust since the security model and algorithms are based on formal bases, scalable thanks to the compositional checking and adaptable to SOA dynamic reconfiguration.

The rest of the paper is structured as follows. Section II describes program dependence graphs as abstractions for BPEL workflows. Section III presents the security model and E2SM configuration synthesis algorithms. Section IV evaluates E2SM mainly the algorithm performance. Section V presents related work and section VI concludes the paper and presents its perspectives.

2 Dependence Graph abstraction

In this section, we describe program dependence graphs (PDG) as abstraction for BPEL processes for data tracking and the system dependence graphs (SDG), our extension to PDG to abstract composed BPEL processes.

2.1 Program Dependence Graphs

Program dependence graphs (PDG) are a standard tool to model information flow through a program [11]. Graph nodes represent program statements or expressions. A data dependence edge, represented with an arrow, $x \rightarrow y$ means that statement x assigns a variable which is used in statement y (without being reassigned underway). A control dependence edge $x \rightarrow y$ means that the mere execution of y depends on the value of the expression x (which is typically a condition in an if- or while-statement). A path $x \rightarrow^* y$ means that information can flow from node x to node y. Contrarily, if no path exists from node x to y, it is guaranteed there is no information flow from x to y. To identify all statements influencing a node y, the backward slice is defined as $BS(y) = \{x \mid x \rightarrow^* y\}$. PDG is classically used in imperative languages like Java [8][11]. We implemented in a previous work [22] a PDG generator for BPEL processes. A BPEL process is composed mainly of two types of activities, that are (1) basic activities, such as receive, reply, invoke, assign, throw, exit, and (2) structured activities, such as sequence, if, while, repeat − until, pick, f low. First, a BPEL control dependence sub-graph is constructed where nodes represent BPEL activities and edges represent possible execution sequences of the activities. This is applicable mainly for condition activities like < if... > or < switch... >. For example, a control flow edge $x \rightarrow y$ means that the activity represented by y may execute immediately after the execution of the activity represented by x. Second, a data dependence analysis is performed attributing the system variables to their activities and based on a Definition-Use relation. For each ordered pair (n_d,n_u), where a statement called n_d contains a definition of a variable v and used in a statement n_u, a data dependence is identified [12]. For example, Fig. 1. illustrates the graphical representation of the BPEL process of a laboratory service deployed on a cloud. It receives a patient medical record and following the record type, it forwards the record to the radio laboratory service or the blood laboratory service. The activities are indexed with their node number.

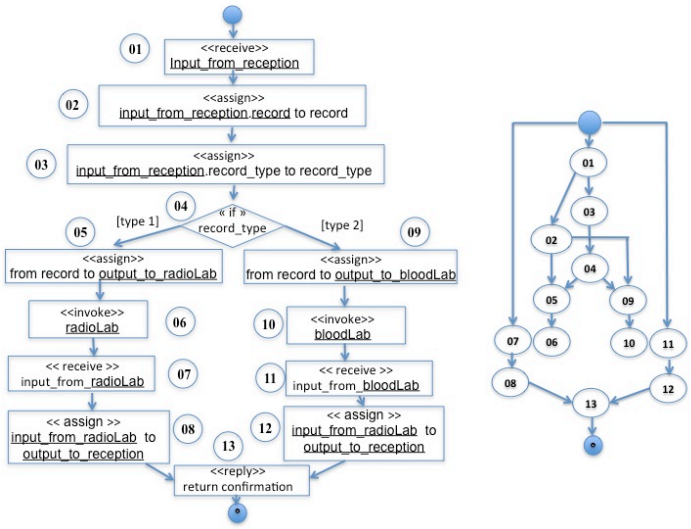

Fig. 1. Graphical representation of the Laboratories BPEL and the correspondent PDG

For example, node (01) corresponds to the receive BPEL activity and node (02) corresponds to the copy instruction of the assign activity. In (01), variable *input_from_reception* is used and in (02) this variable is copied to the newly defined record variable. Since there is a Definition-Use relation between the two activities w.r.t input from reception variable, a graph edge is created between node (01) and (02). Similarly, there is an edge between node(01) and (03). Node (04) corresponds to a structured if activity where variable record type is used. The Definition-Use relation with respect to variable *record_type* allows creating the graph edge between (02) and (04). On the other hand, a control dependence creates the edge between (04) and (05) and between (04) and (05) due to the condition activity at node (04).

2.2 System dependence graphs

For composed BPEL processes, SDG describes the information flow for the entire system. To build it, we consider the PDG of all services and connect them with binding edges. Each binding edge corresponds to an inter-service connection: the source is the sender output endpoint and the sink is the destination service endpoint. Furthermore, for each primitive service, we construct a restricted PDG with nodes corresponding to input and output variables and edges joining the input nodes to the output nodes they depend on. Variable dependence in atomic services can be identified either using classical program dependence graphs [13] or using a description of the service behavior. In the worst case, we consider that all service outputs depend on all service inputs. Hereafter, we give a formal definition of SDG.

Definition 1 (System Dependence Graph). Let WSS be a WS system composed of n services $(S_i)_{i=1,n}$. For each service S_i , let $G_i = (N_i , E_i)$ be its associated PDG. The system dependence graph G = (N,E) is constructed as follows: $N = \cup_{i=1,n} Ni$ and E =

$E_{inter} \cup \cup_{i=1,n} E_i$ where E_{inter} is the set of edges corresponding to inter-service bindings.

3 E2SEM security model

Many security annotation models are proposed in the literature [5]. Nevertheless, DLM, the decentralized label model [14] initially proposed for Java programs, is more appropriate for distributed services with mutual distrust. We extend DLM for SOA and we define non-interference for BPEL-based workflows. Afterwards, we propose two algorithms for non-interference checking and configuration synthesis.

3.1 Decentralized Label Model

The essentials of DLM are objects containing information to protect, principals and labels. Objects can be variables, storage locations or input/output communication channels receiving/sending data. Principals are persons or programs that own or access pieces of system information. To express confidentiality policy, labels are associated to objects and a principal can be owner initiating the information or reader authorized to access that information. A label general structure is $L = \{o_1 : r_{11},r_{12},\cdots ; o_2 : r_{21},r_{22},\cdots ; \cdots ; o_n : r_{n1},r_{n2},\cdots\}$ where o_i are owners and r_{ij} are readers. For example, in a clinical SOA, the medical record can be labeled L = {patient : clinic,doctor} which indicates that the patient principal allows the clinic and the doctor to access his information. Labels are ordered using the no more restrictive than relation, represented by $\subseteq$ symbol. Given two labels L1 and L2, we have $L1 \subseteq L2$ if and only if owners of L1 are included in L1 and, for a given owner, the readers of L2 are included in those of L1. For example, if L1 = {clinic : doctor1} and L2 = {patient : doctor1, doctor2}, we have $L2 \subseteq L1$. During program computation, when information is labeled with L1 and L2, respectively, the result should have the least restrictive label that maintains all the flow restrictions specified by L1 and L2. This least restrictive label, the join of L1 and L2 (written as $L1 \cup L2$), is constructed so that owners is the union of L1 and L2 owners and the reader set for each owner in L1 and L2 is the intersection of their corresponding reader sets. For example, let us suppose that a medical record is composed of two parts so that the first part is labeled L1 = {patient : doctor1} and the second one is labeled L2 = {patient : doctor1,doctor2}. The medical record has to respect both constraints and then only doctor1 is allowed to read the medical record content with label $L1 \cup L2$ = {patient : doctor1}. Note that security labels represent a lattice $(L, \subseteq)$ [14] where L denotes the finite set of labels that is partially ordered by $\subseteq$. We denote by * the weakest principal used to annotate public information. For example, label {p : *} indicates that information is owned by principal p which allows all principals access its information.

Principals can delegate their information ownership to other principals with regard to a *trust* relation. According to trust hierarchy, information can be relabeled in a safe way following two forms. In the first form, label ownership is changed. A label's owner O can be replaced by owner O' so that O *trusts* O'. The second form of relabeling is declassification, which allows relaxing security constraints in a safe way. The information is copied in a fresh variable and relabeled to a less restrictive security

label. This is allowed only by data owner or by another principal trusted by the owner. The new label is usually calculated from the information label by adding readers (more principals are allowed to access information) or removing some policies (owners with their respective readers), which imposes less reading constraints on that information. For example, in the Fig.1. use-case, the patient record labeled L={patient: laboratory} does not allow initially blood-Lab nor radio-Lab to read data. Nevertheless, since these labs are trusted by the patient, they can be added as readers and L is declassified to L2={patient, laboratory, blood-Lab, radio-Lab}.

Since we have two kinds of services in SOA, managed and external services, we extend DLM with **required** and **provided** labels. Required labels are immutable labels that represent external constraints, typically third-party service requirements that cannot be changed by the administrator. Provided labels are labels set on managed resources and can be modified by the administrator when needed.

3.2 PDG-based non-interference checking

Let N be the set of PDG nodes, X the set of program variables and L the set of security labels. Let S: $N \cup X \rightarrow L$ be a function assigning security labels L to nodes in $N \cup X$ defined as follows. For a program statement represented by a node n, if a variable v is defined in that statement, then S(n) = S(v). For example, in Fig.1., we have S((01)) = S(*input_from_reception*). If no variable is defined in the statement represented by the node n, then $S(n) = \cup_i L_i$ where L_i represents the security label of i^{th} variable used in that statement [11]. Non-interference property is satisfied in annotated PDG with S if and only if for every edge $x \rightarrow y$ it holds $S(x) \subseteq S(y)$. Classically, non-interference is iteratively checked starting from specific nodes called slicing criteria that we define hereafter. For a statement of interest represented by a node x, the backward slice BS(x) extracts those statements that potentially have an influence onto that statement. This later is called the slicing criterion. In our work, we define slicing criteria are the nodes that have required labels. Typically, a slicing criterion is a node representing an endpoint to an external service and imposing its security constraint. Consequently, required security label specifies a limit so that only information having a smaller security label may reach that statement. Inversely, provided labels can be assigned to any node in PDG. Formally, provided labels are defined by a partial function P: $N \cup X \rightarrow L \cup \{\perp\}$. Similarly, required security is defined as a partial function R: $N \cup X \rightarrow L \cup \{\perp\}$. In this model, the security label S(n) of every node n must moreover satisfy: $P(n) \subseteq S(n) \subseteq R(n)$ whenever P(n) and/or R(n) are defined. In this extended model, the non-interference property in an annotated PDG can be verified as follows.

Proposition 1 (Non-interference checking - version 1). In a PDG with P and R the functions assigning respectively the provided and required security labels, the non-interference property holds if the following condition is satisfied: $\forall n \in dom(R), \forall x \in dom(P) \cap BS(n), P(x) \subseteq R(n)$. Considering actual security labels S, the next proposition provides an alternative way for checking non-interference [11].

Proposition 2 (Non-interference checking - version 2). In a PDG with P and R the functions assigning respectively provided and required security labels, non-interference property is satisfied if the following condition holds: $\forall n \in dom(R)$, $S(n) \subseteq R(n)$.

Algorithm 1: Secure configuration without declassification
Input: G = (N, →) a PDG graph, X a set of variables,
P, R provided and required security annotations on N ∪ X
Output: S complete label assignment on N ∪ X
/* Initialization */
foreach n ∈ N ∪ X do
  S(n) := {⊥}
  if P(n) *defined* then S(n) := P(n) endif
end
/* Main Loop */
while *(S changes)* do
  foreach n ∈ N do
    S(n) := S(n) ∪ $\cup_{n' \in BS(n)}$ S(n′)
    if *(node* n *defines variable* x*)* then
      S(n) := S(n) ∪ S(x) ; S(x) := S(n) endif
end
foreach n ∈ N ∪ X do
  if R(n) *defined and* S(n) ⊄ R(n)) then
    stop /* required security label exceeded at n */ endif
end

Algorithm 2: Secure configuration with declassification
Input: G=(N,→) a PDG, e the root node,
X a set of variables,
P, R provided and required security annotations on N ∪ X
Output: D, the set of declassified nodes
/* Declassification nodes is initially empty */
D := ∅

/* Main loop */
while (D changes) do
  /* Compute candidates for declassification */
  Z := { z∈N / z defines a variable, ∃x∈dom(P),
      x∈BS({z},D∪{e}), ∃y∈dom(R),
      z∈BS({y},D∪{e}), P(x) ⊄ R(y),
      checkTrust(this, P (x), R(y)) }

/* Interact with user to select a subset Z′ for declassification */
  Z′ := *chooseForDeclassification*(Z)
/* Augment the set of declassified nodes */
  D := D ∪ Z′
end
/* Final check */
  if conditions of Prop 3 hold then
    exit /* secure configuration obtained */

end

Starting from an initial annotation for provided and required security levels R and P, Algorithm 1 calculates the actual security configuration S using an iterative method. Algorithm 1 can be considered as a flavor of the classical Bellman-Ford algorithm. The number of iterations depends on relabeling occurrence which depends itself on the number of vertices, the graph connectivity and the initial label distribution.

Regarding declassification, specific nodes are selected to be declassification nodes [15]. A declassification node d has a security label S(d) and a required security label R(d) so that the relation $S(d) \subseteq R(d)$ is not satisfied. Declassification implies that the user authorizes lowering S(d) to R(d). Non-interference with declassification holds if for each path from node x to y where the relation $S(x) \subseteq S(y)$ is not true, there is a declassification node d on the path with $S(x) \subseteq R(d)$ and $S(d) \subseteq S(y)$ (assuming that there is no other declassification node on that path) [11]. Therefore, information flow control with declassification is no longer transitive. For confidentiality checking with declassification, a simple solution is to represent declassification nodes as barriers where slicing stops [16]. Barrier slices are defined as follows.

Definition 2 (Barrier slice). Let G = (N, E) be a PDG, C a slicing criterion and B the set of barrier nodes. The barrier slice BS(C,B) for the slicing criterion C is the set of nodes on which a node $n \in C$ (transitively) depends via a path that does not contain any node of B [16].

Checking confidentiality with declassification implies checking non-interference considering barrier slices instead of backward slices where the barrier nodes are composed of the declassified nodes and the entry node (corresponding to the entry point in the program). The slicing criterion is composed of the join of nodes with required labels and declassified nodes. Consequently, proposition 1 can be adapted considering slices with barriers as follows.

Proposition 3 (Non-interference with declassification checking). In a PDG with P and R the partial functions assigning respectively the provided and required security labels, D a set of declassified nodes, e the root of the PDG (corresponding to the entry point of the program) and B a barrier node set where $B = D \cup \{e\}$. Non-interference with declassification is satisfied if the following condition holds:

$\forall n \in dom(R) \cup D, \forall x \in dom(P) \cap BS(\{n\}, B), P(x) \subseteq R(n)$

Based on this definition, we propose Algorithm 2 that helps users checking and building secure configurations with declassification. Algorithm 2 calls the *checkTrust*(S, L1 , L2) function that, for each service S and labels L1 and L2, verifies the declassification condition based on trustfulness between principals.

Proposition 4. The algorithm *Secure configuration checking* accepts secure configurations and rejects non-secure ones.

Proof. Let S be the computed configuration. For each node x with required security, we have one of the following conditions: either $S(x) \subseteq R(x)$ or x is a declassified node. Otherwise, the algorithm stops. By definition, we have a secure PDG and then S is a secure configuration.

3.3 SDG-based non-interference checking

To deal with system security, we extend the definition of security configuration defined for single WS (see Definition 1) to WS composition. A security system configuration is an assignment of security labels to variables and nodes within all the PDG of services composing the WS system.

Definition 3 (Security system configuration S). Let WSS be a WS system composed of a set of n services $(Si)i=1,n$. For each service Si , let Gi = (Ni , Ei) be the associated PDG and let Xi the set of its variables. Let Einter be the set of edges corresponding to inter-service bindings. Let L be the set of labels, X the set of all service variables and N the set of all PDG nodes associated to WSS. We define a security configuration for a WS system as a mapping $S : X \cup N \longrightarrow L$ that associates security labels to variables and nodes. Moreover, we require the following three matching conditions amongst the different categories, for all i, j: (1) $\forall v \in X_i, \forall n_V \in G_i$, v defined at $n_V \Rightarrow S(v)=S(n_V)$, (2) $\forall v \in X_i, \forall n_V \in G_i$, v used in $n_V \Rightarrow S(v) \subseteq S(n_V)$ and (3) $\forall n \in G_i, \forall m \in G_j$, $(n,m) \in E_{inter} \Rightarrow S(n)=S(m)$.

The two first conditions correspond to security configuration definition for a single service. The third condition states that the nodes corresponding to binding edges have the same security label since they hold the same information. By analogy to secure PDG, we define now a secure SDG as a connection of secure PDG. For SDG, we have two kinds of barrier slices: internal barrier slices inside PDG joining input variable nodes to output variable nodes and external barrier slices that correspond to inter-service bindings. For internal slices, since PDG are assumed to be secure, we have the guarantee that information does not flow from high level labels to low level labels except for declassification nodes. For external slices, the security enforcement ensures a matching between labels of inter-service nodes connecting PDG. Furthermore, in SDG, there is no room to declassification since it was treated locally to each PDG and for each declassified node ni, its actual security label was assigned to its required one, that is $S(n_i) = R(n_i)$.

Definition 4 (Secure SDG). Let G = (N,E) be an SDG of a WS system and $G_i = (N_i, E_i)$ the set of PDG of services composing the system for $1 \leq i \leq n$. We say that G is secure iff $\forall 1 \leq i \leq n, G_i$ is a secure PDG.

We define an end-to-end secure system as a system where information do not flow from high-level sources to lower level destinations except for special destinations where the user authorizes information declassification. Information dependence is detected thanks to PDG for BPEL activities and for atomic service programs.

Definition 5 (End-to-end secure WS System). Let WSS be a WS system comprising a set of composed WS services and let S the security configuration of the system. We say that WSS is end-to-end secure if for all y, x two variables, y depends on x implies $S(x) \subseteq S(y)$.

Proposition 5. Let WSS be a WS system and G its associated SDG. If G is secure then WSS is end-to-end secure.

Proof. We prove the proposition by induction on the number n of composed services the information flows through. Consider the basic case where n = 1. This means that information flows inside a single composed service. Let us consider any two variables y and x so that y depends on x. That means, there exist a path in the PDG from a node n_x to a node n_y where respectively x and y are defined. Since the PDG is secure, we have $S(n_x) \subseteq S(n_y)$. By matching variable and node labels, we have $S(x) \subseteq S(y)$. Consider now that the proposition is true for any WS system composed of $n-1$ services for some $n > 1$. We prove that it is true for n composed services. Let $S_1, .., S_n$ be the system services, $G_1, .., G_n$ their PDGs supposed secure and consider that information flows from variables x to y, which moreover belong to separate services S_1 and S_n. Let z be the output variable calculated from x inside S_1. Since, G_1 is secure, we have $S(x) \subseteq S(z)$. (1)

Suppose that z is further received by S_2. By the induction hypothesis, the system composed of $n - 1$ services $S_2, .., S_n$ is end-to-end secure. Then $S(z) \subseteq S(y)$. (2)

(1) and (2) implies S(x) ⊆ S(y). As a conclusion, the proposition is true for a WS system with any number n of composed services. As illustrated in Algorithm 1, inside each process, the binding between the process and the service endpoints it communicates with, is checked before checking security inside the process. Applying the previous proposition, this ensures that the system is end-to-end secure.

3.4 Non-interference runtime checking

At run time, non-interference re-checking occurs when a label value changes corresponding to a security constraint modification. The non-interference checking and configuration synthesis is executed for the concerned process. If the label of a binding edge changes, the other edge's label is updated and then the affected service has its security configuration re-checked and synthesized. The same logic is applied when a structural re-configuration occurs in the SOA. Indeed, adding or removing a service to the application implies adding or removing a binding and this may induce modifications to edge labels.

4 E2ESEM evaluation

Scalability is ensured thanks to the composed checking and security configuration synthesis that can be performed in a parallel way. For the algorithm's performance evaluation, we generate a single source graph so that the number of vertices ranges from 5 to 4000 vertices and generates edges in a probabilistic way. We use Erdos-Renyi model to generate edges. So edges are generated with a probability p where p is set to three values: 0.1 for weakly connected graph, 0.5 for moderated connected graph and 1 for full connected graph. Without loss of generality, we consider two security levels : secret and public. We distribute randomly public and private labels on graph edges and do not consider required labels not to stop the program iteration before visiting all edges.

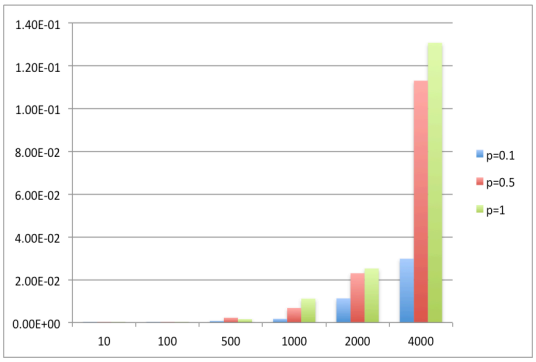

Fig. 2. Performance evaluation of configuration's synthesis program

Fig.2. shows the average time obtained after repeating the program execution 10 times. The bench is executed on a Mac OS version 10.8.5 with processor 1.3 GHz Intel Core i5 and 4GB of Memory. The execution time increases with the number of vertices in the PDG and with the number of connectivity. We clearly see that the execution time is very acceptable and does not exceed 1 second for 4000 vertices and a fully connected PDG.

Adaptability to special cases is ensured thanks to the declassification feature, which

allows a controlled relaxation of security constraints. E2SEM declassification can be extended to other kinds of declassification like temporal declassification [9]. Regarding system's reconfiguration by adding or removing a service, thanks to the compositional verification, there is no need to recheck the whole newly obtained application, only affected processes have their security configuration re-checked and synthesized.

5 Related Work

Our work is related to information flow security solutions for SOA. SEWSEC [17] is an end-to-end security tool for WS security. Compared to SEWSEC, our work allows not only non-interference checking but also security configuration synthesis, provides a formal security model and deals with adaptation. In [18], a security configuration synthesis is provided but the adopted model does not deal with declassification and adaptation. Similarly, in [19], information flow security is applied to component-based systems. Nevertheless, component code is required for label propagation, no formal model is provided and adaptation is not considered. Information flow control is also treated for event-based communications like those described in BPEL [20][21]. For data dependence tracking, systems are modeled as Petri-nets [22] [23] or propagation graphs from workflow's log data [24]. A language based information flow is proposed in [25] to check non-interference but declassification and adaptation are not supported. In [26], [27], authors deal with chained services with no centralized orchestration service composing them. A recent work extends BPEL-orchestration engine [28] and requires BPEL processes' annotation whereas we propose a more practical approach with minimal configuration effort and configuration synthesis.

6 Conclusion

In this paper, we propose a robust security model to protect confidential data in complex business applications. Even though, we concentrate on BPEL, the work is applicable to other types of SOA composition languages. They only need to be mapped to program dependency graphs. We are currently implementing E2SM associated tool and experimenting it to secure e-health real SOA application.

References

1. Walsh, A.E., ed.: Uddi, Soap and Wsdl: The Web Services Specification Reference Book. Prentice Hall Professional Technical Reference (2002)
2. Web services security: Soap message security 1.1. http://docs.oasis- open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf (February 2006)
3. Bajaj, S., al.: Web services policy framework (wspolicy). http://specs.xmlsoap.org/ws/2004/09/policy/ws-policy.pdf (March 2006)
4. Della-Libera, G., al.: Web services security policy language (ws-securitypolicy). http://specs.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.pdf (Jully 2005)
5. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. Proceedings of the 1982 Symposium on Security and Privacy (1982)

6. Zdancewic, S.: Challenges for information-flow security. In: Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence. (2004) 5–19
7. Alves, A., al.: Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf (April 2007)
8. Ferrante,J., Ottenstein,K., Warren,J. : The program dependence graph and its use in optimization. In: ACM Transactions on Programming Languages and Systems. Volume 9. (1987) 319–349
9. Sabelfeld, A., Sands, D. : Dimensions and principles of declassification. Journal of Computer Security (2009) 255–269
10. Myers, A., Sabelfeld, A., Zdancewic, S.: Enforcing robust declassification. Computer Security Foundations Workshop (2004) 172
11. Snelting, G., Robschink, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. ACM Transactions on Software Engineering and Methodology 15(4) (2006) 410–457
12. Mao, C.: Slicing web service-based software. International Conference Service-Oriented Computing and Applications (SOCA) (2010)
13. Giffhorn, D., Hammer, C.: Precise analysis of java programs using joana. (2008) 267–268
14. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol (2000) 410–442
15. Hammer, C., Krinke, J., Nodes, F.: Intransitive noninterference in dependence graphs. Second International Symposium on Leveraging Applications of Formal Methods Verification and Validation (2006) 119–128
16. Krinke, J.: Slicing, chopping, and path conditions with barriers. In: Software Quality Journall. (2004)
17. Zorgati, H., Abdellatif, T.: SEWSEC: A secure web service composer using information flow control. In: 6th International Conference on Risks and Security of Internet and Systems, Timisoara, Romania, IEEE (2011)
18. Ben Said, N., Abdellatif, T., Bensalem, S., Bozga, M.: A robust framework for securing composed web services. In: Formal Aspects of Component Software - 12th International Conference, Brazil. (2015) 105–122
19. Abdellatif, T., Sfaxi, L., Robbana, R., Lakhnech, Y.: Automating information flow control in component-based distributed systems. ACM Sigsoft International Symposium on Component-based System Engineering CBSE, ACM (2011)
20. Bartolini, C., Bertolino, A., Marchetti, E., Parissis, I.: Data flow-based validation of web services compositions: Perspectives and examples. Training (2008) 298–325
21. Bartolini,C.,Bertolino,A.,Marchetti,E.,Parissis,I.:Dataflow-basedvalidationof web services compositions: Perspectives and examples. In: Architecting Dependable Systems. (2008) 44–60
22. Busi, N., Gorrieri, R.: A survey on non-interference with petri nets. Security (2004) 328–344
23. Busi, N., Gorrieri, R.: Structural non-interference, in elementary and trace nets. Mathematical Structures in Computer Science (2009) 1065–1090
24. Accorsi, R., Wonnemann, C.: Static information flow analysis of workflow models. BPSC (2010)
25. Dieter, H., Melanie, V.: Information flow control to secure dynamic web service composition. Control (2006) 196–210
26. Wei, S., I-Ling, Y., Bhavani, T., Elisa, B.: The scifc model for information flow control in web service composition. 2009 IEEE International Conference on Web Services (2009) 1–8
27. She, W., Yen, I.L., Thuraisingham, B.: Enhancing security modeling for web ser- vices using delegation and pass-on. 2008 IEEE International Conference on Web Services (2008) 545–552
28. Demongeot, T., Totel, E., Le Traon, Y.: Preventing data leakage in service orchestration. In: IAS. (2011)