Finding Secure Compositions of Software Services: Towards A Pattern Based Approach

Luca Pino and George Spanoudakis

Abstract—**In service based systems, there is often a need to replace services at runtime as they become either unavailable or they no longer meet required quality or security properties. In such cases, it is often necessary to build compositions of services that can replace a problematic service because no single service with a sufficient match to it can be located. In this paper, we present an approach for building compositions of services that can preserve required security properties. Our approach is based on the use of secure composition patterns which are applied in connection with basic discovery mechanisms to build secure service compositions.**

Index Terms — **software service security, secure service composition**

I. Introduction

In service based systems (SBS) – i.e., systems that provide their functionality by orchestrating external software services outside their own control and ownership – there is often a need to replace services which become unavailable or no longer meet required quality (e.g., performance) or security properties (e.g., confidentiality, availability, privacy) at runtime. In such cases, sometimes it is not possible to find single substitute services for the one which should be replaced. A possible way for addressing such cases is to attempt to build a composition of services that could replace the problematic service.

A fundamental challenge associated with such cases is the need to ensure that all security properties, which are required from an SBS and have to do with the service to be replaced, are taken into account whilst identifying suitable compositions.

Addressing security properties in runtime service composition has received little attention in the literature (e.g., [9][10]) and, to the best of our knowledge, existing work does not provide a comprehensive solution to the problem. The work we present in this paper is aimed at addressing this gap.

Our approach is based on patterns of service composition that are known to preserve certain security properties and can be applied at runtime in order to find secure service compositions. These patterns specify abstract and parametric specifications of service workflows, preconditions for their application and security implication rules. These rules express formally proven logical implications between security properties and, during the composition process, they are used to infer the security properties which would need to be satisfied by individual services in order to guarantee some other security property required of the composition as a whole. Once identified the security properties required of individual services within a composition are fed into a discovery tool which identifies candidate services that satisfy the properties.

The approach that we present in this paper extends a tool that has been developed at City University to support the discovery of services at runtime based on criteria referring to the interface (i.e., inputs and outputs), behaviour and quality properties of services [17]. This tool supports runtime discovery in two modes: a *reactive and a proactive* mode. In the reactive (or "pull") mode, services are discovered only when a need for them arises. In the proactive (or "push") mode, service discovery queries are provided to the tool for each of the constituent services of an SBS. These queries are executed in parallel with the operation of an SBS to identify and maintain up-to-date sets of candidate services that could be used to replace the constituent services of the SBS when any of them fails. The push mode has been shown to improve significantly the efficiency of the discovery process [17]. Service composition should also be applied in push mode as it is computationally expensive and cannot be expected to produce timely results if applied reactively.

The rest of this paper is structured as follows. In Sect. II we present a scenario for service composition which we use in the rest of the paper to exemplify our approach. In Sect. III, we introduce the secure service composition patterns. In Sect. IV, we describe the process of applying these patterns to generate compositions that preserve security properties. Finally, in Sect. V we overview related work and in Sect. VI we provide some concluding remarks.

II. Scenario

To exemplify our approach, in the rest of the paper we use a scenario where a composition of services needs to be built in order to replace a service S acting as a broker for car hire companies and providing quotes for hiring cars. S takes as input the profile of a driver (i.e., a parameter of type Person) and produces a personalized offer based on car hire companies that are within a given distance of the driver's address. A security property regarding S is that the confidentiality of any input information passed to it must be preserved as it contains

Luca Pino is with City University London, Northampton Square, London EC1V 0HB, UK (corresponding author; email: Luca.Pino.1@city.ac.uk).

George Spanoudakis is with City University London, Northampton Square, London EC1V 0HB, UK (email: G.E.Spanoudakis@city.ac.uk).

This work was supported by the EU F7 grant Assert4SOA (grant 257351).

personal driver information (i.e., his/her full name, day of birth and address).

Our approach assumes that in an SBS using service S designers must have specified a query Q_S that can be executed to find a replacement of S if S becomes unavailable or malfunctions at runtime. At runtime, if after executing Q_S there is no single service matching Q_S's conditions, a composition that could replace S should be found. An example of a composition in this case could involve two services: *CompanyLoc* and *BrowseOffer*. In this composition, *CompanyLoc* is a car hire company locator service taking as input the address of person, who needs to hire the car, and returns the closest car hire companies to that address. *BrowseOffer* gets quotes for hiring cars from the located companies and enables their browsing.

To preserve the confidentiality of driver data, in this example, all the data that is passed to *CompanyLoc* as well as any information that is derived from these data and would be passed from *CompanyLoc* to *BrowseOffer* and/or stored by any of these services should remain confidential.

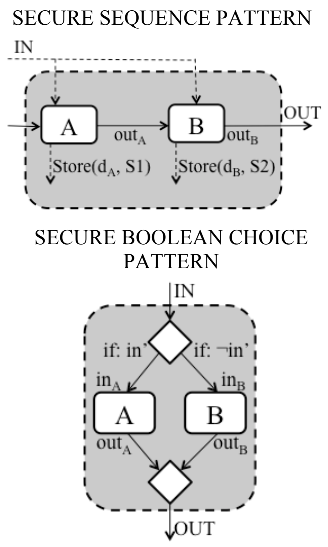
III. Secure Composition Patterns

Our approach uses *secure composition patterns* to drive the process of constructing secure workflows of services. A secure composition pattern specifies an abstract elementary workflow of *activities*, that should be provided by individual services (or further compositions of services), and the *control* and *data flows* connecting these activities. An example of an elementary workflow is the sequential workflow in which services are composed in chains and invoked sequentially to achieve the required functionality outcome [1][2].

In addition to the actual workflow, a secure composition pattern specifies: (a) the overall *security property(ies)* that a workflow can ensure subject to the security properties of the services bound to the individual activities in it, (b) *suitability conditions* that must be satisfied in order to apply the workflow, and (c) *dependencies* between the inputs and outputs of the different activities of the workflow (referred to as *IO dependencies* in the following).

Two examples of secure composition patterns are shown in Fig. 1. The first of these patterns is the *Secure Sequence Pattern (SSP)*. This pattern contains two abstract activities, namely A and B, and control flows between them (shown as solid arrowed lines) indicating that activity A should be executed and completed before B. The pattern specifies also data flows showing that the inputs of the service that the pattern may be used to replace (IN) may be used only partially by A (in_A) and B (in_B). The IO dependencies specified for SSP indicate that the inputs passed to activity A should be a subset of IN ($in_A \subseteq IN$), the inputs to activity B should be a subset of IN and the output of A ($in_B \subseteq IN+out_A$) and the output of the final activity of the pattern (B) should be a superset of the outputs that the pattern will replace ($OUT \subseteq out_B$).

Security properties are represented as relations over patterns or services and the data consumed, produced, or stored by them (i.e., the input, output, and persistent internal data, respectively). Our approach uses a built-in but extensible ontology of relations of the form *<relation>(<service/pattern>, <data>)* in order to express security properties. The confidentiality of the inputs and outputs of the sequential pattern is expressed, for example, by the relations *conf(self, IN)* and *conf(self,OUT)*, respectively.



Security Properties
Conf(self,IN), Conf(self, OUT)
IO dependencies for A
input: $in_A \subseteq IN$
IO dependencies for B
input: $in_B \subseteq IN+out_A$
output: $OUT \subseteq out_B$

Suitability conditions
$\exists$ *in'* $\subseteq$ *IN:*
if (in') then execute(A)
else execute(B)
Security Properties
Conf(self,IN), Conf(self, OUT)
IO dependencies for A
input: $in_A \subseteq IN$
output: $OUT \subseteq out_A$
IO dependencies for B
input: $in_B \subseteq IN$
output: $OUT \subseteq out_B$

Fig. 1. Examples of secure composition patterns

The security properties of a secure composition pattern are used in order to check the applicability of the pattern during the service composition process. In particular, the patterns are retrieved and applied during the search for a composition (see Sect. IV) only if the security properties that they satisfy match with the security properties specified in the discovery query of the service that the composition is to replace. Following the selection of a secure composition pattern, during the application of the pattern, its IO dependencies are used as querying conditions for discovering services that could match the individual activities in the pattern.

The ability of a pattern to guarantee the security properties included in its specification depends on the actions that the activities of the pattern may perform on the data handled by the pattern. To ensure the confidentiality of input data in the sequential pattern, for example, it is necessary to ensure: (a) the confidentiality of the data transmitted from A to B (i.e., out_A), and (b) the confidentiality of the data stored by A (i.e., d_A) if out_A and d_A may disclose information about in_A (and consequently IN) and A derives any of these data from in_A. To express such conditions regarding the preservation of security properties, it is necessary to specify actions that pattern activities or the individual services which instantiate them perform on data. These actions are specified by the so called *security-related actions*.

Table I lists the different types of actions that may affect security. As shown in the table an activity/service may: project a given set of data D over some of the properties of these data (*Project(D,D')*); select a subset of data D' from D without affecting their structure of the original data (*Select(D,D')*); derive data D' from a given set of data D in a way that discloses information about a specific set of properties {P} of D (*Derive(D,D',{P})*): store or alter D data in a persistent store (*Store(D,L)* and *Alter(D,L)*); and/or retrieve D data from a persistent store (*Retrieve(D,L))*

TABLE I
SECURITY-RELATED ACTIONS ON DATA

Action	Explanation
Project(D, D')	Subgraph(Type(D'),Type(D))
Select(D, D')	Extension(D') ⊆ Extension(D)
Derive(D, D', {P})	D' is derived from D in a way disclosing info about the set of properties {P} in the type of D
Store(D, L)	Data D are kept in persistent store L
Retrieve(D, L)	Data D are retrieved from persistent store L
Alter(D, L)	Data D are altered in persistent store L

In some cases the actions performed on data are specified for the individual activities at the pattern level. In the sequential pattern of Fig. 1, for example, it is specified that activities A and B store data (see *Store*(d_A,S_1) and *Store* (d_B,S_2)). In other cases, however, the pattern itself might not specify conditions about the actions that its abstract activities may perform on data.

As discussed earlier, the ability of a pattern to guarantee certain security properties depends on the actions that the individual services, which will be bound to the activities of the pattern, perform on the data and the security properties that hold for these individual services. These dependencies are expressed by *security implication rules*.

As an example, consider the confidentiality property. A security implication rule should express that, if the data D, used in an activity A as an input or an output, is derived from another data D', that was required to be confidential for some action A', and the derivation of D from D' can disclose some property of D', then the confidentiality of D in A is required.

Security implication rules are specified as part of composition patterns only if their validity is formally proven. Proofs of security implication rules must have been constructed offline prior to the publication of a secure composition pattern and the use of the pattern in the service discovery and composition process. This is necessary as any attempt to construct proofs of rules at runtime, or equivalently, derive dependencies between security properties of aggregate workflows and individual workflow services from first principles is computationally expensive and, thus, impractical to do over and over again every time that the discovery process tries to instantiate a specific part of a workflow. Hence, we assume that the security implication rules have been formally proven before becoming part of the specification of a pattern and can therefore be safely used to infer the security properties of individual services when the pattern is applied. The process of constructing proofs of security implication rules is beyond the scope of this paper but interested readers may find examples of such proofs in [4].

To specify the security implication rules in patterns we use *Situation Calculus (SC)* [5]. SC is a first order logic (FOL) language introduced originally to model and reason about dynamical domains. In particular, SC may be viewed as a dialect of FOL, were the predicates that can have different truth values are called fluents. Each fluent is evaluated against a specific sequence of actions passed to them, called *situation*.

The specification of security implication rules assumes that secure composition patterns and the information of the instantiated services in the workflow are also expressed in SC[1]. In particular, we use

- the fluent *next(A,A')* to specify that an activity A is followed by an activity A' in the workflow of a pattern
- the fluent *input(A,D) (output(A,D))* to specify that D is an input (output) of activity A
- the *security-related actions* of services
- the fluent *known(P, S)* to specify that the security property P is already known to be satisfied (certified) in situation S.

In this model the situations are the different traces of the workflow, and *currAct(A)* is the valid fluent when the reasoning step is on activity A. The reasoner walks through the workflow and at each step it is possible to check which *security properties* are required through the fluent *requires(P,S)*.

TABLE II
EXAMPLE OF SECURITY IMPLICATION RULES

PRECONDITION AXIOMS
$poss(step(A),S) \leftrightarrow currAct(A',S) \wedge next(A,A')$
SUCCESSOR STATE
$currAct(A,do(\alpha,S)) \leftrightarrow \alpha = step(A)$
$requires(conf(A,D),do(\alpha,S)) \leftrightarrow [\alpha = step(A) \wedge (input(A,D) \vee output(A,D)) \wedge known(conf(A',D'),S) \wedge derive(D',D,\varepsilon) \wedge \varepsilon \neq \varnothing \wedge A' \neq A] \vee [\alpha \neq step(A) \wedge requires(conf(A,D),S)]$

Table II shows the specification in SC of the security implication rule that we introduce informally above. The first two rules are general rules that specify how the situation evolves. The actual rule for the confidentiality property is the third one, and it basically follows the explanation from before (the only addition is the part after the disjunction and it is a common solution to the frame problem in SC).

Whilst applying a *security composition pattern*, the activities in the pattern should be instantiated based not only in finding services that match with the IO dependencies of the pattern but also with security conditions that may be associated with the particular activity. The exact security conditions required for each activity are inferred from: (a) the *security properties* that the composition which is being built by the pattern must satisfy, (b) the set of services that have been already bound to activities of the pattern and the *actions* that these services perform on data, and (c) the *security implication rules*.

Security implication rules state which security properties would be required of the individual activities within a pattern in order to guarantee that a given *security property* of the composition defined by the pattern as a whole will also hold.

The process of deriving the security properties that should hold for the individual services that can be bound to a pattern is discussed in Sect. IV below.

[1] This assumption is not restrictive since a high level, even graphical specification of patterns can be translated to the SC representation that we use for specifying and reasoning with security implication rules.

IV. Composition Process

The composition process focuses on building workflows through the application of the *secure composition patterns*. This process starts when no single replacement service has been found for a service S that needs to be discovered for an SBS and it is initially driven by the same discovery query (Q_S) that has been specified by the SBS designers to drive the single service discovery for S. More specifically, initially, the security conditions are collected from Q_S and the secure composition patterns that are known to guarantee security properties satisfying these conditions are retrieved. If no such pattern is found, then no replacement can be found for S.

For each of the retrieved patterns, the process tries to find services that could be bound to each of the activities of the pattern. The search for candidate services for each activity may start from the initial or final activity of the pattern. Once an appropriate service for this activity is found the pattern is partially instantiated by binding the located service to the activity and then searching for bindings to the activities that are neighbors of the one of the instantiated activities. If more than one candidate services are found for the current activity, a different instantiation of the pattern is created by binding each of these services to the current activity and the composition process continues by considering each of these alternative instantiations.

The security conditions required of the candidate services for an activity during the search are determined after finding the candidate services based on all the conditions of Q_S except those related to security. This is because security conditions may also depend on the actions that the services that will be bound to an activity perform, as we discussed in Sect. III.

In the case of SSP, for example, if the confidentiality of the inputs IN to the workflow that will be created by the pattern is required, and a candidate service S for the activity A in the workflow is known to derive its output data (out_A) from the inputs passed to it (in_A) in a way that discloses information about these data (i.e., it is known that S performs the action *Derive*(in_S,out_S,{*Type*(in_S)}), then two separate security properties will be required of S: (i) the confidentiality of its input data (*conf*(S,in_S)) and (ii) the confidentiality of output data (*conf*(S,out_S)). The second condition would not, however, be required if S did not derive its outputs from its inputs or the derivation of its outputs did not disclose any information about in_S. Note that in this example, without regard to security conditions, to be a valid candidate service for A, S would have to store some data d_S to an internal data store L_S. Thus, if d_S was also known to be derived from the inputs of S (i.e., it was known that S performs the actions *Derive*(in_S,d_S,{*Type*(in_S)}) and *Store*(d_S,L_S) then the confidentiality of the persistent data of S, i.e. (*conf*(S,D)), would also be required for S to be a valid candidate for A.

During the composition process, the exact security conditions to be checked for a candidate service of an activity are derived from the *IO dependencies* of the activity and the actions that the service performs, using the security implication rules of the pattern being applied. The check of whether the service complies with the security properties required from it is based on security certificates, created by some independent authority and published in a service registry [16]. Such certificates also confirm the actions that a service performs on data.

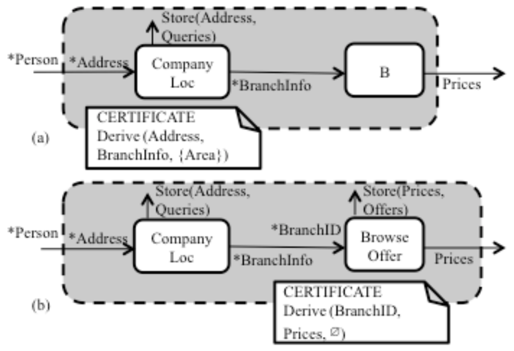


Fig. 3. Example of progressive pattern instantiation. Data marked with (*) must be confidential according to security implication rules.

As an example of the application of this process consider the example introduced in Sect. II and the generation of a composition to replace the car hire quotation service S. A pattern that can guarantee the confidentiality of the driver profile that will be given as input to this composition is the SSP pattern. The query to instantiate the first activity in SSP will then just require a service whose input is a subset of the input provided to S or, equivalently, a service with an input data type which is a supertype of the input data type of S.

Suppose that the discovery for the first activity of SSP returns the service *CompanyLoc*. The activity A of SSP is then instantiated with this service (see Fig. 3(a)), if *CompanyLoc* complies with the security conditions derived for it from the pattern. In particular the initial security condition for S was that *Person* should be confidential. Thus, the *security implication rules* will infer that the confidentiality for *CompanyLoc*'s input *Address* is also required, as the latter is a projection of *Person*. Furthermore, since the output of *CompanyLoc*, namely *BranchInfo*, is derived from the driver's *Address* data and can potentially reveal information about the *Area* part of the address, *BranchInfo* will also be required to be confidential. This inference would be made by the rule of Table II, assuming that the action *Derive(Address, BranchInfo, {Area})* is specified in a certificate within the description of *CompanyLoc*.

Subsequently, the query for the second activity of SSP (B) is built. The *IO dependencies* of SSP require that the input of B is a subset of *Person+BranchInfo*. The security properties required for the replacement of S and the *security implication rules* would then require that *Person* and *BranchInfo* must be confidential.

Suppose that a service called *BrowseOffer*, requiring just *BranchID* from *BranchInfo*, is located. The security implication rules can then infer that *BranchID* should be confidential (as a projection of *BranchInfo*). As shown in Fig. 3(b), the *prices* that *BrowseOffer* outputs and stores in a persistent store don't need to be confidential since as indicated by the certificate of *BrowseOffer*, prices are derived from *BranchID* but don't disclose any information about it.

If the search for a service for a specific activity in a partially instantiated pattern fails, the composition process attempts to apply some composition pattern recursively in order to find a composition of services that could be bound to the activity of concern. This recursion may generate complex workflows as indicated in Fig. 4.

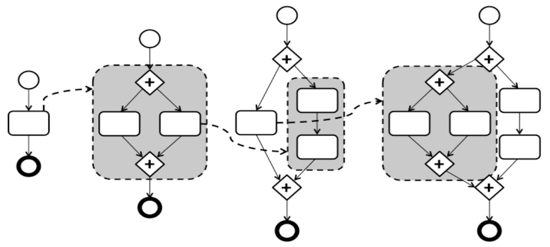


Fig. 4. Recursive application of composition patterns.

V. Related Work

Research dealing with security in service composition has focused on the verification of the security of existing compositions through model checking [6][7][8]. Our focus, however, is different since we are looking into applying composition patterns that are proven to guarantee security properties as part of a runtime service discovery and composition process.

A work that is more related to ours is [9], where planning techniques are used to compose workflows that are compliant with some lattice-based access control models (e.g. multi-level secure systems). The focus of [9] is how to find efficient algorithms for sequential workflow planning whilst our approach is more general w.r.t both the types of workflows and the security properties that it covers.

In [10] the authors describe an approach to security conscious web service composition through matching security constraints required for service provision and constraints declared by service providers. The security constraints in this approach are specified in SAML [11]. In [10], secure service compositions are generated based upon some pre-defined domain specific business workflows, whilst our approach allows the generation of arbitrary workflows.

Other works on automatic service composition (e.g. [2][12][13][14]) allow the expression of security properties in discovery queries, usually as non-functional properties. These approaches focus on specific types of security properties and check them only against single services in compositions, without addressing the overall security of a composition.

Aniketos project [15] also uses secure composition patterns (i.e. sets of rules) and checks them against existing composition plans. Aniketos patterns describe service configurations leading to either secure or insecure situations, and are used after the composition process, to check if a required security policy applies. In our work the security check is performed during the composition process driving it.

Finally, our secure service composition patterns are similar to the workflow patterns in [3] as they specify elementary workflows that can be used to generate service compositions. However, our patterns include additional applicability and security properties.

VI. Conclusion

In this paper, we have presented an approach supporting the identification of secure service compositions, as part of runtime service discovery. Our approach is based on secure composition patterns. These patterns specify abstract service workflows, and the security properties that they are known to preserve if their constituent services have certain security properties. The logical connections between service and composition level security properties are expressed by security implication rules and the reasoning for deriving the former properties from the latter is based on modeling patterns, properties and security implication rules in Situation Calculus.

Our work builds upon an existing runtime service discovery framework [17] and extends it with secure service composition capabilities.

Currently, we are investigating the use of standardized languages, notably SAML, for expressing security properties and the development of an ontology to express properties at several granularity levels and dependencies between them.

References

[1] A. Zisman, K. Mahbub and G. Spanoudakis, “A service discovery framework based on linear composition,” in *Proc. IEEE Int. Service Computing Conference (SCC 2007)*, pp.536-543, 2007

[2] F. Lécué, E. Silva and L. F. Pires, “A framework for dynamic web services composition,” in *Proc. 2nd ECOWS Work. on Emerging Web Services Technology (WEWST07)*, 2007.

[3] W. M. P. Van Der Aalst et al., “Workflow patterns,” *Distrib. Parallel Databases* 12(1): 5-51, 2003.

[4] ASSERT4SOA Project, “D5.1– Formal models and model composition,” 2011.

[5] J. McCarthy and P. Hayes, “Some philosophical problems from the standpoint of artificial intelligence,” in *B. Meltzer and D. Michie, editors, Machine Intelligence*, 4:463–502, 1969.

[6] M. Deubler, J. Grünbauer, J. Jürjens and G. Wimmel, “Sound development of secure service-based systems,” in *Proc. of the 2nd Int. Conf. on Service oriented computing (ICSOC '04)*, pp. 115-124, 2004.

[7] Jing Dong, Tu Peng and Yajing Zhao, “Automated verification of security pattern compositions,” *Inf. Softw. Techn.* 52(3):274- 295, 2010.

[8] M. Bartoletti, P. Degano and G. L. Ferrari, “Enforcing secure service composition,” *18th Work. on Computer Security Foundations,* 2005.

[9] M. Lelarge, Z. Liu and A. Riabov, “Automatic composition of secure workflows,” in *Proc. of ATC'2006*, 2006.

[10] B. Carminati, et al., “Security conscious web service composition,” in *Proc. of the Int. Conf. on Web Services (ICWS)*, 2006.

[11] OASIS. SAML 1.0 Specification Set [Online]. Available: http://saml.xml.org/saml-specifications, 2002.

[12] Keita Fujii and Tatsuya Suda, “Semantics-based dynamic web service composition,” *IEEE Journal on Selected Areas in Communications*, 23(12): 2361- 2372, Dec. 2005.

[13] B. Medjahed, A. Bouguettaya and A. K. Elmagarmid, “Composing web services on the semantic web,” *The VLDB Journal*,12(4):333-351, 2003.

[14] M. C. Jaeger, G. Rojec-Goldmann and G. Muhl, “QoS aggregation for web service composition using workflow patterns,” in *Proc. of the 8th Int. Conf. on Enterprise Distributed Object Computing,* 2004.

[15] Aniketos Consortium, “D3.1 – Design-time support techniques for secure composition and adaptation,” [Online]. Available: http://www.aniketos.eu/, 2011.

[16] M. Bezzi, A. Sabetta, Spanoudakis G. “An architecture for certification-aware service discovery, ” *1st Int. Work. on Securing Services on the Cloud*, 2011

[17] A. Zisman, G. Spanoudakis, J. Dooley. “A framework for dynamic service discovery”, In *Proc. of 23rd Int. ACM/IEEE Conf. on Automated Software Engineering, 2008*