

# Deriving Systems Level Security Properties of Component Based Composite Systems

Khaled M. Khan  
School of Computing and  
Information Technology  
University of Western Sydney  
Locked bag 1797, S. Penrith D.C.  
NSW 1797 Australia  
k.khan@uws.edu.au

Jun Han  
Faculty of Information and  
Communication Technologies  
Swinburne University of Technology  
PO Box 218, Hawthorn, Melbourne  
Vic 3122 Australia  
jhan@it.swin.edu.au

## Abstract

*This paper proposes an approach of defining systems-level security properties of component-based composite systems. It argues that the security properties of a composite system can be viewed either from the end-users' point of view, or from the software integrators' point of view. End users look more for the ultimate security goals achieved in the composite system, whereas software integrators are more interested in the compositional security properties of the system in terms of the required and ensured properties. Software integrators need to know how a composite system could be assembled further as a coarse-grained component with other applications. It is equally important for the end user of the system to know the actual security objectives achieved at the systems-level.*

## 1. Motivation

In component-based software engineering, an application system is composed of several stand-alone software components developed by third parties. These components are readily available from various distributed sources for runtime execution. However, the paramount security concerns of these 'third-party' software components and their compositions with large scale systems over the Internet have become a major challenge. In a highly fluid distributed environment, software integrators are virtually forced to compose systems with third-party components of which they have only partial or no knowledge about their underlying security properties. When components are acquired from the Internet and composed with an application system, it is unclear what type of ultimate security can be achieved with these components for the enclosing system.

The entire composite system from time to time may require different types of functionality and accordingly, the composition of the security contracts between components needs to be re-characterised to match the re-configuration requirements of the system in order to support systems security evolution. In a component-based system, we need to achieve both the security compatibility between interacting components and the security objectives for the entire system. When the entire composite system is running and providing meaningful services to the users, the systems-level security properties need to be characterised.

To illustrate the main focus of this paper, we consider the following systems scenario. A number of individual healthcare components provide independent services to their environments. The services include keeping patients information, providing diagnosis reports based on test data, offering specialists advice based on the diagnosis reports, quoting prices for prescribed medicines and so on. Each of these services are independent and catered by different individual stand-alone components. We can compose a complete healthcare system by integrating the individual services provided by different components. Such a composite system is considered as a collaboration of different components which provide services in a seamless and secure way. These service providing individual components deal with a whole range of sensitive issues related to patient information such as diagnostic reports, MRI/CT-Scan images, pathological test results, diagnosis reports and prescriptions. For example, a component offering pathological services may advertise its services as providing confidentiality through secure storage of test results. On behalf of its patients, another component offering general medical services (e.g., GP practices) may require such confidentiality provided at a particular level (as guaranteed by specific encryption schemes with specific key lengths). The pathol-

ogy component may or may not satisfy the general practitioner's (GP) requirements, depending on how the confidentiality guarantee is realised. How do we automatically define and characterise the ultimate security achievement of the entire composite system assembled with this type of different components with diverse security requirements and assurances? Once individual components with their own security provisions are assembled together to form a composite system, what would be the actual security property of that enclosed system? Current component based technologies do not offer any ready solutions to such advanced, and yet obvious requirements for security-aware systems composition.

Based on our approaches reported in [5, 6], in this paper we attempt to characterise the systems-level security of the entire composite system. A systems-level security contract (SsC) is characterised on the basis of the ultimate functionalities that a composite system offers to the external world. An SsC exposes the security properties of the composite system that are visible to the external world.

This paper is organised as follows. Section 2 outlines our earlier work on techniques of security characterisation which is used in this paper to characterise systems-level security properties. Section 3 discusses the systems-level security goals from users' perspective and the systems-level security properties from software integrators' perspective. A discussion on the related work is cited in section 4. Finally, we conclude in section 5.

## 2. Security characterisation: an overview

First we very briefly describe the fundamentals of our security characterisation language support [5] which will enable readers to familiarise the terminology we use in this paper. A component can be internally composed of several primitive components. In order to achieve a particular functionality, a component may require to be composed with other components at the lower level. We refer them *component-level* in the system hierarchy. Components composed at the lower level may not be perceived at the *systems-level*. Figure 1 shows such compositions at three different levels of the abstraction: *component-level*, *compositional-level* and *systems-level*. A compositional-level is a composition between two components to achieve a partial or full functionality.

The figure shows that, at the systems-level, an application called *calculate salary()* is composed of three components. The entire system has two compositional-level and one systems-level compositions. One of these two compositional-levels is established between *get\_net\_salary* and *get\_gross\_salary*. The other one is made between *get\_net\_salary* and *print\_slip*. The dotted straight lines refer to the *component-level* compositions which are inter-

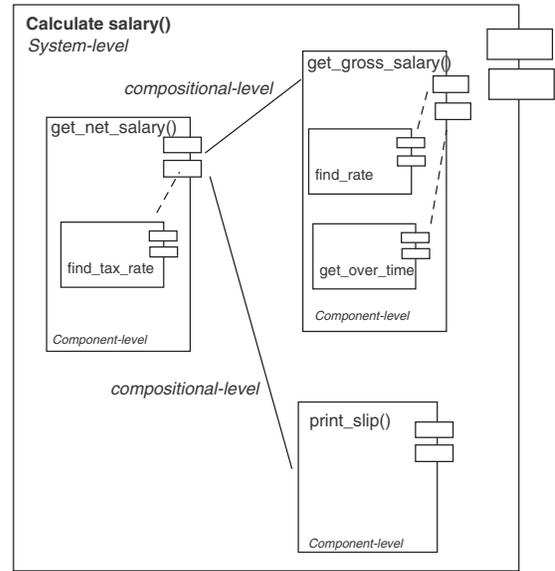


Figure 1. Hierarchy of system composition

nal to the component. The component *get\_gross\_salary* has a composition with two internal components *find\_rate* and *get\_over\_time* at the *component-level*. The component *get\_net\_salary* has one *component-level* composition with the component *find\_tax\_rate*.

The security properties achieved at a compositional-level between two components are referred to compositional security contracts (CsC). In other words, a CsC is based on the compatibility between the required security properties of a component and the ensured security properties of another components [7]. A CsC is always established between two participating components satisfying each others security requirements. At the systems-level, a security contract of a composite system is based on several CsCs. We call such security properties as systems-level security contracts (SsC). Several CsCs contribute to the establishment of an SsC [6].

A system can be configured and reconfigured incrementally as components are added and removed. Hence, the configuration of the composite application changes accordingly. In a composite system, the security properties of one component may be consumed by other participating components. However, a particular type of security property at the component-level may be considered quite differently at the systems-level. For instance, a security property protecting an external communication channel between two components at the component-level could be regarded as protecting internal communication at the systems-level (at least logically). In this paper,

we focus our discussion entirely on the systems-level security properties.

To facilitate the automatic security-aware composition, a language support is required that could sufficiently express the security properties of the components so that the systems integrator, as well as the other components, could identify the suitable components and reason about their security compatibility. To automate the security characterisation and to reason about the compatibility of compositional security contracts, our language support [5] is based on *logic programming* and *BAN logic* [3].

In this language support, security properties are represented and expressed in symbolic notations called *atoms*. An *atom* consists of a predicate name (a security function) with a tuple of *variables* or *constants* (elements). Atoms usually express relationships between elements. An atom is of the form  $p(t_1, \dots, t_n)$  where  $p$  is an  $n$ -ary predicate symbol and  $t_1, \dots, t_n$  are terms [14]. A term could be either a variable, a constant, or a function representing an element such as an entity, data, a password, a key. *Variables* are used to generalise objects such as  $X$  for which the inference rules find or substitute an element.

The predicate name of an atom represents a security property such as *encrypted*, *key\_generated* and *signed*. An example of an atom is  $encrypted(p, x)$ . It states that an element identified as  $p$  encrypts the object  $x$ . The entity  $p$  could be a human, a program or a component. The object  $x$  could be a message, a piece of data, or a file. Another example of an atom is  $signed(p, x, k)$ . It states that the entity  $p$  digitally signs the object  $x$  with the key  $k$ .

In addition to the representation of security properties and their associated elements, inference rules are used to check the conformity of the security properties represented in atoms. A clause is composed of a head and a body. A rule is of the form  $r \leftarrow l_1, \dots, l_m, not\ l_{m+1}, \dots, not\ l_n$ , where  $r$  is the head of the rule (an atom) and the literals  $l_1, \dots, l_n$  ( $n \geq 0$ ) constitute the body of the rule [14]. The symbol ( $\leftarrow$ ) is read as ‘derives’. The left hand side of  $\leftarrow$  (the head of the rule) is the conclusion or consequence that represents the ensured security properties. An ensured security property is a conclusion of an expression. The right hand side of  $\leftarrow$  (the body of the rule) is called a premise or condition [1]. It represents the required security properties in our characterisation context. Examples of such rules are:

$$\begin{aligned} signed(tax, k^{-y}) &\leftarrow encrypt(form, k^y). \\ CsC(x, y) &\leftarrow signed(tax, k^{-y}). \end{aligned}$$

In the above expressions, the left hand side of the  $\leftarrow$  is the ensured property and the right hand side properties are required properties. The first rule states that the object  $tax$  is digitally signed with the private key

$k^{-y}$  of  $y$  if the object  $form$  is encrypted with the public key  $k^y$  of  $y$ . Note that the terms  $k^{-y}$  and  $k^y$  represent private and public keys of the component  $y$  respectively. The second rule states that a compositional security contract (CsC) between  $x$  and  $y$  is established if  $tax$  is digitally signed by  $y$  with its private key  $k^{-y}$ .

Rules make an inference with the knowledge associated with the security properties. If each atom in the body of a rule is *true*, then the inference rule concludes a *true* to the head. Rules make an inference with the security facts of the problem domain. A set of inference rules are applied to prove whether a CsC is achievable or not.

Based on these fundamentals, we present the characterisation of systems-level security of composite systems in the following sections.

### 3. Security properties of composite systems

The entire composite system is viewed as a coarse-grained component. An SsC of a composite system, therefore, can be viewed from two related perspectives: (1) the *end-users perspective*, the ultimate *security goals* of which are achieved from a particular functionality provided by a collection of components; and (2) the *software integrators' perspective*, that is, the ultimate *required and ensured security properties* of the composite system for further composition. The former one is the high level security objective achieved in a composite system, whereas, the latter perspective concerns with how the high level security objectives are achieved in terms of low level derived security properties of the composite system. The relationship between these two perspectives is crucial to understand the security of the composite system.

#### 3.1. SsC from end-user's perspective

Once the composition of a component based system is complete, the end-users as well as the systems integrators would be interested to know what security objective is achieved at the composite systems-level. End-users may not be very interested in the abstract format of the security property specified at the systems-level. A security goal is the ultimate security objective of a security property. For example, a security goal could be defined as the integrity of an object, the confidentiality of a message, an authentication of an entity, the authorisation for an operation on certain object, or a nonrepudiation of a message. In an end-users perspective, the ultimate security goals that are actually achieved in a particular functionality may be provided by a collection of atomic components in a composition. The security goals of the composite system expose the ultimate security objectives achieved

in the system. Let us first consider an example. Assume that a CsC is established between two components such as:

Example (1).

$$CsC(john, bank) \leftarrow encrypted(amount, k), signed(amount, k^{-bank}).$$

In this example, the rule specifies that a CsC is established between two entities *john* and *bank* if the *amount* is encrypted with the public key *k* and it is digitally signed by the private key  $k^{-bank}$  of the *bank*. All the required security properties are internal to the participating components. The term  $k^{-bank}$  denotes the private key of the *bank*. The reasoning of this CsC depends on the required properties. However, for an SsC, such detailed information about the security properties may not be always required at the end-user level. Users may rather know which security goals are likely to be achieved in a particular functionality. This example does not state what type of security goal is actually achieved in the composition  $CsC(john, bank)$ . We argue that each concluding composition (i.e. a CsC) should specify at least one underlying security goal. In SsC, we need to spell out the underlying security goals embedded in the CsC rules. In Example (1), we can specify several security goals for the rules. Firstly, the *confidentiality* of the object *amount* is achieved with the atom  $encrypt(amount, k)$ ; and secondly, the *authenticity* of the entity *bank* is established with the property  $signed(bank, k^{-bank})$ . A *nonrepudiation* security goal is achieved from this atom too. We now redefine Example (1) as:

Example (2).

$$CsC(john, bank) \leftarrow encrypted(amount, k), signed(amount, k^{-bank}).$$

$$\begin{aligned} confidentiality(amount) &\leftarrow encrypted(amount, k). \\ authenticity(bank) &\leftarrow signed(amount, k^{-bank}). \\ nonrepudiation(bank) &\leftarrow signed(amount, k^{-bank}). \end{aligned}$$

The last three rules in Example (2) state the ultimate security goals achieved by this CsC. The atom  $confidentiality(amount)$  specifies that the object *amount* is confidential while being transferred between the entities *john* and *bank* because it is encrypted. Similarly, the atoms  $authenticity(bank)$  and  $nonrepudiation(bank)$  specify the authenticity and nonrepudiation of the entity *bank* respectively because the bank digitally signed the amount with its private key.

From the end-user's perspective, a security goal describes the ultimate security behaviour that users of the system can experience from their direct interaction with the enclosed system. An achieved security goal is very much dependent on how the entire application is perceived and used by the end-users in terms of systems func-

tionality.

To facilitate understanding of the composite security properties in terms of their goals, we propose a format as:

$$goal(X) \leftarrow security\_property(X, Y, Z)$$

which specifies the security goal of the term *X* in relation to the specified security property. The *X* is a term that is used as an element in the atom participated in establishing a CsC between two entities. The CsC is established based on this term. However, it might have more other terms associated with the CsC. For example, a CsC is established if a *certificate* is digitally signed. In this case, the security goal would be the objective which is achieved for the *certificate* with the signature. The goal could be authenticity or the nonrepudiation of the *signer*. The predicate *goal* could be defined with any of the following: (1) *confidentiality*: the secrecy of an object or message; (2) *integrity*: the untampered value of an object; (3) *availability*: the authorised entities that have access to an object, service, or operation; (4) *authenticity*: confirms the identity of an entity; (5) *nonrepudiation*: the undeniable evidence of actions performed by an entity; (6) *auditability*: the auditing and monitoring security events; and (7) others.

The term *X* could be either an object or an entity. A security property might have more terms, and hence may have multiple security goals. The term *X* must be a part of the atom involved in a CsC rule. A goal must reflect the security achievement of the security property involved. Note also that a security property may have more than one security goal. Consequently, the rule to identify the security goal is: (1) assign a security goal to each type of security property, (2) the term used in the atom must be used as a term in one of the CsC rules.

For our security characterisation purposes, we have codified the security functional properties and security assurance properties defined in the ISO standard, Common Criteria (CC) for Information Technology Security Evaluation [13] and mapped them onto the security goals. Table 1 shows some examples of the security properties and the corresponding goals. The functional requirements defined in Common Criteria (CC) provide a schema for evaluating the IT system and enumerate the specific security requirements for such systems. It describes the security behaviour or functions expected of an IT system to withstand threats.

### 3.2. An example of security goals

Consider the following functionality of a composite system: Assume a composite system identified as *printRandom* is composed of two independent components: *random* and *print*. The *print* component makes a request for generat-

Security properties	Security goals
$shared(K, P, Q)$	Confidentiality
$encrypted(X, K)$	Confidentiality
$signed(X, K)$	Authenticity, Nonrepudiation
$key\_exchanged(K, P, Q)$	Confidentiality

**Table 1. Security properties mapped onto security goals**

ing a series of random numbers to the component *random*. The functionality of component *random* generates a series of random numbers to its environment. Assume a compositional security contract is established between the components *print* and *random* on the basis of the following compositional rule:

$$CsC(print, random) \leftarrow signed(random\_nr, k^{-1}), owned(k^{-1}, random).$$

It states that a CsC is established between the components *print* and *random* if the generated random numbers are digitally signed with a private key which is owned by the component *random*. The term  $k^{-1}$  denotes the private key of an entity. In this example, this private key is owned by the component *random* as stated in the second term  $owned(k^{-1}, random)$ . The security properties of the component *random* are stated as

Component::*random*:

$$signed(random\_nr, k^{-1}) \leftarrow . \\ owned(k^{-1}, random) \leftarrow .$$

The properties of *random* ensure that the generated random numbers are digitally signed with the private key of *random*. The security properties of the component *print* are:

Component::*print*:

$$CsC(print, random) \leftarrow signed(random\_nr, k^{-1}), owned(k^{-1}, random).$$

The CsC rule defined in the component *print* states that a compositional security is established between *print* and *random* if the random numbers are digitally signed by the private key of *random*. The user of this composite system between these components is not aware of these internal rules and security properties of the participating components, but might be interested in the ultimate security goals achieved from the underlying CsC, that is  $CsC(print, random)$ . In this CsC, the security goals could be the authenticity and nonrepudiation of the component *random* as suggested in Table 1. Based on this, we derive the following security goals:

$$authenticity(random) \leftarrow CsC(print, random). \\ nonrepudiation(random) \leftarrow CsC(print, random).$$

The authenticity and nonrepudiation goals are achieved from the property  $signed(random\_nr, k^{-1})$  ensured by *random*. Since the object *random\_nr* is signed by the entity *random*, the component *print* can verify the authenticity of *random* with the corresponding public key. This establishes the security goal authenticity. Further more, the entity *random* cannot deny that it produced the object *random\_nr* to the entity *print*. This predicate, therefore, establishes a nonrepudiation security goal. In other words, the users of the functionality know that the random numbers they received from the component *random* are from an authenticated entity which cannot later deny that it had generated the numbers.

### 3.3. SsC from software integrators perspective

In this perspective, we characterise a composite system exactly the same way as we do for atomic components. In fact, a global federated system can be viewed as a coarse-grained component. Software composers perceive such a federated system not as a collection of components but as a single component. The security characterisation of such a ‘composite component’ can also be specified with the same structure and principles defined in [5, 6] for the characterisation of atomic components’ security. Software composers can use such a ‘composite system’ as a single component with their application. Therefore, we need to find how the ultimate required and ensured security properties of a ‘composite system’ could be formulated out of the underlying components. Again such a characterisation is very much dependent on the functionality provided by the composite system.

Some security properties specified at the compositional-level may be specified differently at the systems-level. For example, a property that was considered external in a CsC could be viewed as internal in SsC. In Table 2, we list some security properties which could be used differently in SsC.

Compositional-level (CsC)	System-level (SsC)
$believe\_sees(P, Q, X)$	$sees(P, X)$
$imported(X, P)$	$intern\_transfer(X, P, Q)$
$exported(X, P)$	$intern\_transfer(X, P, Q)$
$key\_distribute(K, A, S)$	$key\_generated(K)$

**Table 2. Properties at the CsC and SsC levels**

Table 2 shows two types of entry: the property definition in CsC and the corresponding SsC property. A property in CsC needs to be changed to a different property in SsC. It is possible that properties at the component-level may take a different form at the systems-level. For instance, in a composite system, data export-import at the *compositional-level* could be viewed as the internal transfer of data between two entities in a *systems-level* composition (at least logically). We have already discussed the issue of the hierarchy of composition in Section 2.

The compositional security properties of a composite system, in terms of the required and ensured properties, can be modelled as:

$$G_{C_1, \dots, C_j}^i \xrightarrow{f_n^i} (E_{f_n^i} \leftarrow R_{f_n^i})$$

where,  $i$  is the identity of a particular system  $G$  which is composed of a set of components  $C$  identified with the subscript  $1, \dots, j$ . The functionality  $f$ , identified with the subscript  $n$ , has the ensured security property  $E$  and the required security property  $R$ . A composite system can have more than one functionality each of which may have more than one required and ensured properties. This can be further modelled as:

$$\begin{aligned} G_{C_1, \dots, C_j}^i &= \{f_1^i, \dots, f_n^i\} \\ f_n^i &= (R_n^i, E_n^i) \\ R_j^i &= \{r_{j,1}^i, \dots, r_{j,k}^i\} \\ E_j^i &= \{e_{j,1}^i, \dots, e_{j,l}^i\} \end{aligned}$$

where,  $G$  is a composite system with an identity  $i$ ;  $f$  is the functionality of the composite system  $G^i$ ;  $E$  and  $R$  are the ensured and required security properties of the functionality  $f_n$ , respectively. The subscript  $n$  is the identity of a functionality, and the identities of the required and ensured security properties are shown with the subscript  $k$  and  $l$  respectively. Identifying the  $R$ s and  $E$ s of the composite system can be a problem because the configuration of the composite system changes as components are assembled and disassembled with the system. In order to derive the correct required and ensured properties of the composite system, we propose the following algorithms based on our earlier work [6]. Figure 2 indicates the algorithms used to compute the SsC. The description of each algorithm is given below.

1. Identify the functionality  $f_i$  of the composite system  $S$  in which the user is currently interested.
2. Identify all CsCs (compositional security contracts between two components)  $CsC_{[1, \dots, j]}$  involved in the functionality  $f_i$  of the composite system  $S$ .
3. Identify all required security properties  $R_{[1, \dots, m]}^i$  of all

```

(1)  $f_i = \text{determine}(\text{Functionality}, S)$ ;
(2)  $CsC_{[1, \dots, j]}^i = \text{identify}(\text{all\_CsC}, f_i)$ ;
(3)  $R_{[1, \dots, m]}^i = \text{find\_R\_All}(CsC_{[1, \dots, j]}^i)$ ;
(4)  $SsC\_E_{[1, \dots, n]}^i = \text{find\_All\_E}(CsC_{[1, \dots, j]}^i)$ ;
(5) FOR  $k = 1 \rightarrow m$  DO
     $SsC\_R_{[k]}^i = \text{NOT\_Matched}(R_{[k]}^i, SsC\_E_{[1, \dots, n]}^i)$ ;
(6) FOR  $i = 1 \rightarrow m$  DO
    IF  $\text{lookUp}(R_{[i]}^i, SsC\_table) == \text{TRUE}$ 
         $SsC\_R_{[i]}^i = \text{change}(SsC\_R_{[i]}^i, SsC\_table_{[entry]})$ ;
    ELSE  $SsC\_R_{[i]}^i = R_{[i]}^i$ ;
(7) FOR  $j = 1 \rightarrow n$  DO
    IF  $\text{lookUp}(SsC\_E_{[j]}^i, SsC\_table) == \text{TRUE}$ 
         $SsC\_E_{[j]}^i = \text{change}(SsC\_E_{[j]}^i, SsC\_table_{[entry]})$ ;
(8)  $SsC^i = \text{list}(SsC\_E_{[1, \dots, n]}^i, SsC\_R_{[1, \dots, m]}^i)$ 

```

**Figure 2. Algorithms to compute SsC**

the components that have participated in  $CsC_{[1, \dots, j]}^i$  for the functionality  $f_i$ .

4. Identify all ensured properties  $SsC\_E_{[1, \dots, n]}^i$  of the components that have participated in  $CsC_{[1, \dots, j]}^i$  for the functionality  $f_i$ .
5. Identify the required properties  $SsC\_R_{[1, \dots, k]}^i$  from  $R_{[1, \dots, m]}^i$  which need to be satisfied by entities other than the participating components in the composite system  $S$  in order to accomplish the functionality  $f_i$ .
6. Check if any of the identified required properties has an entry in the SsC table. If it does, replace the property with the corresponding SsC entry.
7. Check if any of the identified ensured properties has an entry in the SsC table. If it does, replace the property with the corresponding SsC entry.
8. Finally, list all identified required properties  $SsC\_R_{[1, \dots, m]}^i$  and  $SsC\_E_{[1, \dots, n]}^i$  as the required and ensured security properties respectively of the functionality  $f_i$  of the composite system  $S$ .

We apply these algorithms to an example in order to illustrate their applicability.

### 3.4. An example of SsC

We extend the functionality of our example cited in section 3.2. Assume a composite system called  $PsN$  is composed of three components: *random*, *sort*, and *print*. The functionality of the composite system  $PsN$  is *printSortedRandomNumbers(NUMBERS)*. The composed system

prints a series of random numbers in ascending order. After receiving the random numbers from *random*, the component *print* passes these to the component *sort*. The *sort* component sorts them in ascending order and returns the sorted numbers to *print*. The component *print* finally sends these to the printer. The security properties of these three components are described as follows:

Component::*random*:

$exported\_object(random\_nr, random) \leftarrow$   
 $sees\_user\_attribute(random, User).$   
 $signed(random\_nr, k^{-1}).$   
 $owned(k^{-1}, random).$

Component::*sort*:

$\leftarrow signed(random\_nr, k^{-1}).$   
 $\leftarrow owned(k^{-1}, random).$   
 $\leftarrow encrypted(random\_nr, K).$   
 $\leftarrow owned(K, sort).$

Component::*print*:

$owned(K, sort).$   
 $encrypted(random\_nr, K) \leftarrow$   
 $parts\_of(User\_login, Password).$

Now we identify the required and ensured security properties of the composite system which offers the functionality *printSortedRandomNumbers(NUMBERS)* using the proposed algorithms.

1. The identified functionality of the composite system *PsN* is *printSortedRandomNumbers(NUMBERS)*.
2. The identities of the participating CsCs are *CsC(print, random)*, and *CsC(print, sort)*.
3. From these CsCs related to the functionality, we identify the following required properties of the participating components *print*, *random*, and *sort* which are:
  - $\leftarrow sees\_user\_attribute(random, User),$
  - $\leftarrow signed(random\_nr, k^{-1}),$
  - $\leftarrow owned(k^{-1}, random),$
  - $\leftarrow owned(K, sort),$
  - $\leftarrow encrypted(random\_nr, K),$
  - $\leftarrow parts\_of(User\_login, Password).$
4. We identify all the ensured properties of the components participated in the CsCs. These are: *exported\_object(random\_nr, random)*, *signed(random\_nr, k<sup>-1</sup>)*, *owned(k<sup>-1</sup>, random)*, *owned(K, sort)* and *encrypted(random\_nr, K)*. They are the ultimate ensured properties of the composite system.
5. The required property *sees\_user\_attribute(random, User)* of *random* needs to be satisfied by entities other than the other two participating components because none of the ensured properties of the other two participating

components satisfy *sees\_user\_attribute(random, User)*. *sees\_user\_attribute(random, User)* is the required property of the system. The other required property is *parts\_of(User\_login, Password)* which also needs to be ensured by the user. This property is not ensured by any of the participating components either. All other remaining required properties identified in item 3 are ensured by the participating components.

6. We check if these two required properties have any entry in the SsC table, and assume there is none.
7. We also check if any of the ensured properties has any entry in the SsC table (see Table 2). The property *exported\_object(random\_nr, random)* has an entry. We replace this with the corresponding SsC property with *internal\_transfer(random\_nr, random, print)*.
8. We now list the above identified properties of the composite system as follows:

The ensured properties are:

*internal\_transfer(random\_nr, random, print)*,  
*signed(random\_nr, k<sup>-1</sup>)*,  
*owned(k<sup>-1</sup>, random)*,  
*owned(K, sort)*,  
*encrypted'(random\_nr, K)*.

Note that the *internal\_transfer* property has replaced the property *exported\_object* at the *systems-level*. It is necessary because a component in isolation may consider an object is exported to other component, however, once the component is composed with other components this could be considered an internal transfer at the systems-level. The required properties are:

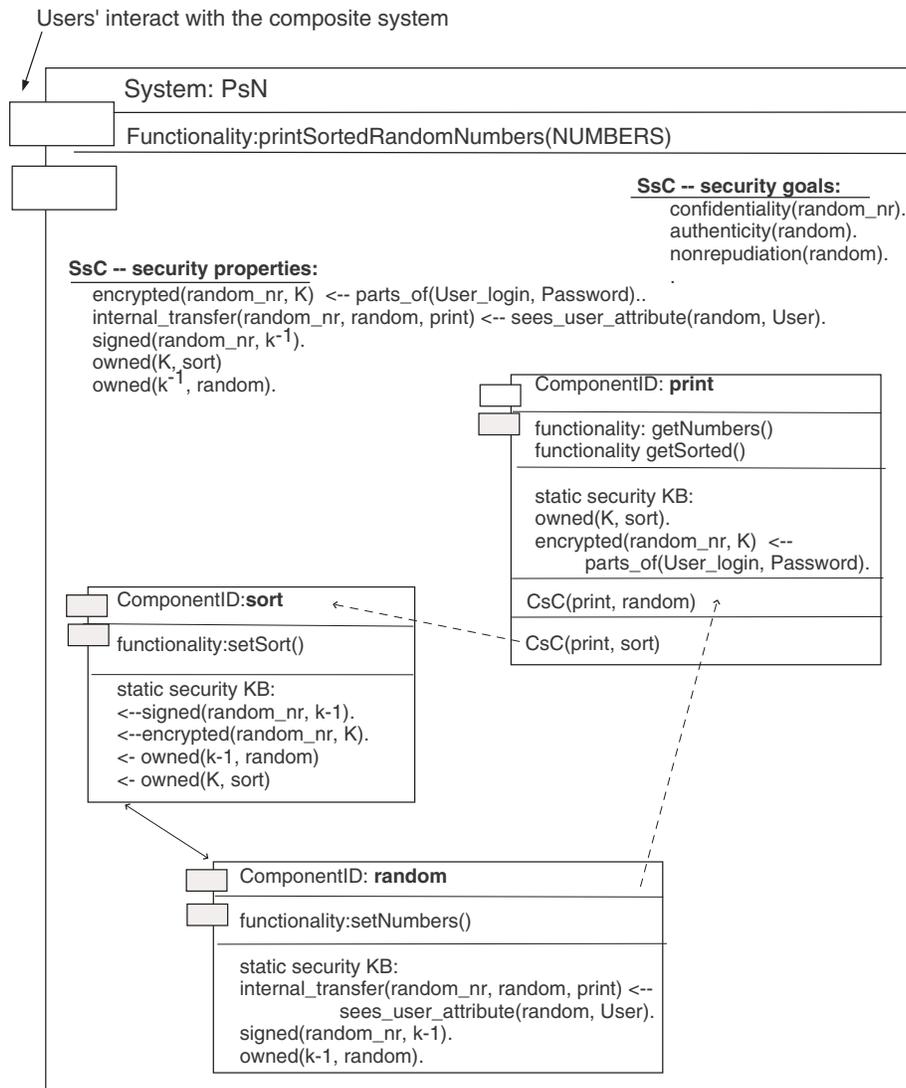
*sees\_user\_attribute(random, User)*.  
*parts\_of(User\_login, Password)*.

Thus, the functionality of the composite system and its security properties are modelled as:

$f_{printSortedRandomNumber}^{PsN} =$   
 $encrypted(random\_nr, K) \leftarrow$   
 $parts\_of(User\_login, Password).$   
 $internal\_transfer(random\_nr, random, print) \leftarrow$   
 $sees\_user\_attribute(random, User).$   
 $signed(random\_nr, k^{-1}).$   
 $owned(k^{-1}, random).$   
 $owned(K, sort).$

More specifically, the require security properties of the composite system are:

*parts\_of(User\_login, Password)*, and  
*sees\_user\_attribute(random, User)*.



**Figure 3. Entire composite system at the user level**

If any system or a component intends to compose with this composite system *PsN* for the functionality *printSortedRandomNumbers(RANDOM)*, it needs to satisfy these two properties. The ensured properties of this composite system are:

*encrypted(random\_nr, K),*  
*internal\_transfer(random\_nr, random, print),*  
*owned(k<sup>-1</sup>, random),*  
*signed(random\_nr, k<sup>-1</sup>).*

Figure 3 illustrates the security properties and the entire composition. The user of the composite system provides

the required properties *parts\_of(User\_login, Password)* and *sees\_user\_attribute(random, User)*. The property *parts\_of(User\_login, Password)* is required by the component *print*, but the property *sees\_user\_attribute(random, User)* is required by the component *random* at the component-level. The user is not aware of which component requires which security property. In return, the system as a whole provides four ensured properties to the user.

In actual terms, the properties *encrypted(random\_nr, K)* and *owned(K, sort)* are ensured by the component *print*, the property *internal\_transfer(random\_nr, random, print)*,

$signed(random\_nr, k^{-1})$  and  $owned(k^{-1}, random)$  are ensured by the component  $random$ . Arrows with dotted lines in Figure 3 show the CsCs, and the solid line with double arrows show the transitive relationship between  $sort$  and  $random$ . The security goals of these security properties of the composite could be spelled out as
 
$$confidentiality(random\_nr) \leftarrow encrypted(random\_nr, K).$$

$$authenticity(random) \leftarrow signed(random\_nr, k^{-1}), owned(k^{-1}, random).$$

$$nonrepudiation(random) \leftarrow signed(random\_nr, k^{-1}), owned(k^{-1}, random).$$

The first security objective ‘*confidentiality* of the random number’ is achieved with the encrypting the random numbers. The second and third security objectives ‘*authenticity*’ and ‘*nonrepudiation*’ of the entity  $random$  respectively are ensured with signing the random numbers by the component  $random$ .

#### 4. Related work

The commercial distributed frameworks such as CORBA, DCOM, .Net and Enterprise JavaBeans (EJB) are used for effective composition between objects or components. What is lacking in those technologies is an automatic mechanism for specifying the compositional specification of the security properties of the participating parties. Although CORBA and EJB form the basis of many functional aspects of the component model, these tools support only some limited sets of non-functional properties such as persistence and access control. CORBA message specification provides a Quality of Service (QoS) framework to define QoS policies for CORBA application [11]. These do not provide any mechanisms for composing non-functional properties along with the functional composition. Attempts have been made to add OMG’s profile for performance, schedulability, and other QoS to the UML specifications [10].

Enterprise JavaBeans (EJB) defines a standard for component composition [17]. The introspection mechanisms provide the compositional information of Bean components such as how other Bean components can be structurally connected with a candidate Bean component. In fact, JavaBeans is not the ideal component model for dynamic security composition in a distributed environment such as Internet.

The emergence of aspect oriented-programming (AOP) [8] explicitly provides a separation of the crosscutting concerns of the system from other functional aspects of the system. The paradigm provides means for systematic identification, representation, and separation of various aspects of the systems. It is accomplished using the

approach called aspect weaver. Different aspects of the system are programmed and then woven together to create a complete executable code. Even different levels of security concerns could be separated in AOP. The work reported in [18] illustrates the use of AOP to programme security by extending C programming. It argues that such an extension could be well used to separate the security concerns from the application code.

The composable security application (CSA) [15] in the Infrastructure for Composability At Runtime of Internet Services (ICARIS) [9] is a Jini and JavaBeans based implementation deploying point-to-point security associations in order to introduce security services to the application. The security association can be built based on the demands of the client and server for a particular security association. However, it requires human assistance. It does not facilitate automatic dynamic negotiation for a security association.

The use of Semantic Web technology based on DAML [4] is a promising technology, although it is still in its infancy. In this regard, the KAoS Policy Ontologies (KOP) [2, 16] designed to analyse policies for agent environments is heavily based on DAML description-logic-based ontology of the computational environment. KAoS services allow enforcement of policies, reasoning of actions based on policies and resolution of conflicts. This work is not security specific, however, the idea could be used to analyse and reason about the security policies.

Not surprisingly, WS-Security addresses Web service security by using existing security standards and specifications. WS-Security is simply a framework that adds existing mechanisms such as XML Encryption, XML Signature and XML Canonicalization into a SOAP message. XML Canonicalization prepares the XML messages to be signed and encrypted. In other words, WS-Security is a specification for an XML-based security metadata container [12]. What WS-Security actually does is that it enables security properties to be defined independently in the message in terms of composable message elements to exchange SOAP messages securely. Similarly, WS-Trust, WS-SecureConversation, Security Assertion Markup Language (SAML) are not much different than the WS-Security. None of these, in fact, introduces a new technique or approach to address security composition for distributed applications.

#### 5. Conclusion

This paper has discussed the systems-level security properties of a composite system from two perspectives. At the end-user level, security goals associated with different security properties are explored. From the software integra-

tor's point of view, we showed how the systems-level security properties could be constructed. This paper illustrates different security goals with a running example. Component developers need to specify the security goals associated with each of the CsC rules. They also need to spell out which security goals are actually achieved. A security goal is ultimately based on the detailed internal security properties (ensured and required) of the components and their compositional properties. We demonstrated with examples that a composite system could also have its own required and ensured security properties such that other component could be assembled with it. In a global federated system, each component, having its own security characteristics, preserves its independence in the compositional contracts and maintains its security property independently, but works in tandem with other components.

## References

- [1] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.
- [2] J. M. Bradshaw, editor. *Software Agents*, chapter KAoS: Towards an Industrial-strength Generic Agent Architecture, pages 375–418. AAAI Press/MIT Press, 1997.
- [3] M. Burrows, M. Abadi, and N. R. A Logic of Authentication. *ACM Transactions Computer Systems*, 8:18–36, February 1990.
- [4] DAML-S. Semantic Markup for Web Services. Technical report, DAML, 2003.
- [5] K. Khan and J. Han. A Security Characterisation Framework for Trustworthy Component Based Software Systems. In *Proceedings of the COMPSAC 2003*. IEEE Computer Society Press, 2003.
- [6] K. Khan and J. Han. A Process Framework for Characterising Security Properties of Component-based Software Systems. In *Proceedings of the Australian Software Engineering Conference*. IEEE Computer Society Press, 2004.
- [7] K. Khan, J. Han, and Y. Zheng. A Framework for an Active Interface to Characterise Compositional Security Contracts of Software Components. In *Proceedings of the Australian Software Engineering Conference*, pages 117–126. IEEE Computer Society Press, 2001.
- [8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP*. Springer-Verlag, 1997.
- [9] D. W. Mennie. *An Architecture to Support Dynamic Composition of Service Components and its Applicability to Internet Security*. M. Eng. thesis, Carleton University, Ottawa, 2000.
- [10] OMG. UML Profile for Schedulability, Performance and Time. Specification version 1.0, Object Management Group, 2003.
- [11] D. Schmidt and S. Vinoski. Object Interactions: Overview of OMG CORBA Messaging QoS Framework. *C++ Magazine*, 2000.
- [12] S. Seely. Understanding WS-Security. Technical report, Microsoft, 2002.
- [13] Standard. Common Criteria for Information Technology Security Evaluation, ISO/IEC 15408. Technical Report v2.0, Nat'l Institute Standards and Technology, Washington, 1999.
- [14] T. Syrjaanen. Implementation of local grounding for logic programs with stable model semantics. Technical report 18, Helsinki University of Technology, October 1998.
- [15] V. Tasic, D. Mennie, and B. Pagurek. On Dynamic Service Composition and Its Applicability. In *Proceedings of the Workshop on Object-Oriented Business Solutions at ECOOP 2001*, pages 95–108, 2001.
- [16] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, M. Bunch, S. Johnson, J. Kulkarni, and J. Lott. KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement. In *Proceedings IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pages 93–98. IEEE Computer Society Press, 2003.
- [17] M. V. and M. Hapner. Enterprise JavaBeans Specification. Technical Report Version 1.0, Sun Microsystems Inc., March 1998.
- [18] J. Viega, J. Bloch, and P. Chandra. Aspect-Oriented Programming to Security. *Cutter IT Journal*, February 2001.