A Process Framework for Characterising Security Properties of Component-Based Software Systems

Khaled M. Khan
School of Computing and IT
University of Western Sydney
Locked bag 1796 S. Penrith DC
NSW 1797 Australia
k.khan@uws.edu.au

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

Abstract

This paper presents a security characterisation process framework for software components and their compositions in order to address the issue of trust in component based software. The process is based on the idea of publishing trust related properties of software components in machine readable as well as understandable form at the component level and incorporating such properties with runtime composition of the application system. We explore the actual process involved in specifying publishable security properties of atomic components, component certification, reasoning about compositional security contracts, and characterising ultimate systems-level security properties to inspire trust in software components.

1. Motivation

Research efforts are currently being carried out mostly in defining component models and compositions for component-based systems. A structural composition consists of components (blackbox entities that export and import functionality), glue code, architectural style, formalisation of component interface and compositional rules [7]. Current frameworks for software component model such as EJB, CORBA, COM, and .Net are limited to the specification of structural interface definition and matching of interfaces [6]. By contrast, the issue of trust in software components has received less attention. In current practice, what a user knows is the component's interface structure on how to connect the component with the user's system and how to get the functionality that the component offers. When a component is available on the Internet, it is unclear which level of trust should be placed in the component because its trust related properties are often unknown to its environment.

In component based software engineering, software integrators suffer greatly from a lack of trust in third-party software components. Trust requirements for software integrators –the frontline component consumers– are quite different from those for the end-users of an application system. The foremost requirement of any trust would be self-disclosure of identity, origin, and security properties that a component has. However, other non-technical factors have tremendous influences in forming trust as well. In this endeavour, we focus on only those factors that are related to the technical aspects of distributed software products.

The current practice of 'uninformed' consent-based trust may work fine at the application level, but at the component level this would unlikely satisfy software integrators. To give such a consent during a dynamic composition to a software component without knowing much about the trust related information might be risky. The current practice may lead software integrators to a situation where they might give consent on components they should not trust, or decline to use those which they should trust.

Consider the following analogy. Whenever an automobile workshop acquires some third-party automobile parts, it usually verifies the specification of the parts, the security and safety instructions if relevant, and other specifications such as the manufacturer details, the standard of the parts and so on. Likewise, when a software integrator acquires a piece of interesting software component from the Internet or other sources for her application, she would naturally look for similar specifications about the component. In most cases she would probably fail to trace any trust related information about the component. If she is lucky enough, in some cases, the component may *claim* if it is secure or not. A universal shallow commitment such as '*the component is secure*' cannot contribute much to in-

Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)
1530-0803/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

spire integrators' trust. A software engineer needs to be ensured with more than this.

An integrator of a component based system is responsible for the design and architecture of the entire application in which a component might play an insignificant role. Whatever role a component plays, software engineers cannot rule out the possible negative impact of that component on their entire application system. Therefore, it is quite natural for them to reason about how the trust related attributes of third-party components would eventually influence the entire system architecture, test plan, and the composition of the application.

Despite the problem with trust in software components, software engineers are still inclined to use third-party software component to minimise the development effort and time. They may hesitate, if not disapprove, to use third-party components by weighing the benefits over the lack of trust in components. The hesitation is primarily based not on the speculation made about the quality of the components, rather largely on the fear of lack of trust related properties of the components such as security, identity, and origin. Software engineers are reluctant to place their complete trust in a component that does not tell much about its trust related properties.

In this background and context, we propose a security characterisation process framework to inspire trust in software components. A security characterisation is based on the principle of disclosure of security properties, origin and identity of components. A security characterisation is a mechanism to support an '*informed consent*' based composition. An informed consent gives the participating entities explicit opportunity to consent or decline whether they use the components or not. If a well defined security characterisation is in place, software integrators are certainly better positioned to evaluate the strength of the security efforts provided by the component in a particular use context. The security characterisation framework not only exposes the publishable security properties to its environment, but it also explicitly provides a reasoning mechanism to integrators for assessing the compatibility of those observable security properties between neighbouring components. The remainder of this paper is dedicated to exploring the actual process involved in these insights.

This paper is organised as follows. Section 2 highlights the major features of our proposed process framework for security characterisation. It also presents an example artefact model of a simple component based system resulting from the proposed framework. Section 3 discusses further the major phases of the framework such as the security characterisation and certification for atomic components, compositional security contracts, and systems-level security properties. The related work is discussed in section 4. In section 5, we draw some conclusions.

2. Security characterisation framework

Our framework includes four aspects of security characterisation: (1) security characterisation of atomic components; (2) certification of characterised security properties at the atomic component level; (3) compatibility checking of security properties between components; and (4) security characterisation of the composite system [8]. Figure 1 shows the component life cycle and the corresponding four aspects of the security charcaterisation process. The life cycle of software components is divided into four distinct phases at different development stages. Firstly, *component development phase* is the pre-certification phase falls under the construction stage. In this phase, security properties of the component's functionality are identified and expressed with the component interface. Secondly, *component packaging phase* concerns with the certification process of the published security properties of the component at the certification stage. Thirdly, *system compositional phase* falls under the deployment stage. In this stage, static or run-time compositional security contracts between components and other systems take place. Finally, in the *post compositional phase*, the ultimate security properties of the federated system is derived based on the security properties of the atomic components and their compositional contracts with others within the system. We call it operational stage because the component is in actual operation at this stage. The low level tasks involved in each of these is further discussed in Section 3.

In Figure 1, the dotted rounded rectangles depict four processes corresponding the four life cycle phases. The arrows between different processes show the sequence and their dependency. Iteration between *characterising security properties* and *approving and sealing published properties* is needed because a component may modify its published properties in a later stage of its life cycle. In that case, it requires a new certificate. Similarly, iteration is required between *compositional security contract* and *systems level security contract* due to constant configuration and reconfiguration of the entire system with various components. The enclosing system from time to time may require different types of functionality and accordingly, the composition of the security contracts need to be re-tested and recharacterised to match the re-configuration requirements of the system in order to support system evolution. The horizontal lines show the separation of processes. The dotted vertical line shows the progression of the processes matching the life cycle of the component in terms of time.

The first two processes of *characterising security prop-*

IEEE COMPUTER SOCIETY

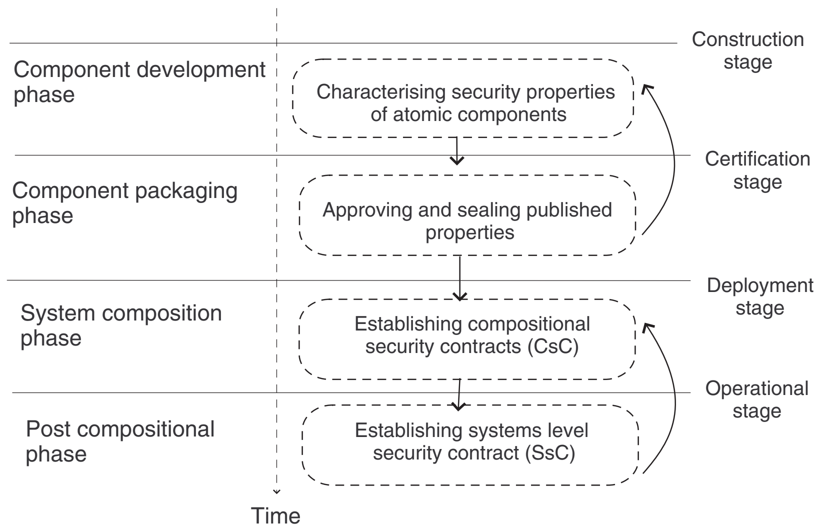

Figure 1. A security characterisation process framework

erties of atomic components and *approving and sealing published properties* are achieved during the development of components, whereas the compatibility checking of the characterised properties is accomplished automatically at the development stage by the interested parties. The characterisation of a composite system is formed when a system is complete at the operational stage. The characterised properties of the atomic components are static in a sense that these cannot be modified unless a change of the component is done, whereas the compositional security contract (CsC) and systems-level security contract (SsC) are dynamic because these properties depend on the actual composition.

In effect, the output of the entire process framework is the artefact model of a component based system where the compositions are established between components based on the conformity of their security properties. Figure 2 illustrates an example of such an artefact model in which compositions are established based on the compatibility of security properties. Three components are identified as x, y and z in this artefact model. The rectangle pairs are the interface symbols of the components. Component x is the focal component and others are candidate components in this system. The terms 'focal' and 'candidate' are used to model the role that a component plays in a particular compositional contract. A software component interested in a service provided by other software components is called a *focal component*. A component providing the requested service to a focal component is called a *candidate component* [5]. In this particular example, component y provides a functionality called $setTax()$, whereas component z offers a functionality called $setClaim()$. On the contrary, component x only receives services from other components in terms of its two functionalities, namely $getTax()$ and $getClaim()$.

A component may provide several functionalities –each may have quite different types of security properties. For an integrator it is hard to predict what security properties a versatile component supports unless these are well expressed. Security properties are used for various reasons such as to authenticate components, to authorise their operation, to ensure confidentiality and integrity, and so on. Examples of security properties can be passwords, private keys, secret keys, public keys, shared keys, digital signatures among others. For the characterisation purposes, we classify security properties into two different types: *ensured security properties*; and *required security properties*. We discuss these further in section 3. The correlation between various system properties of Figure 2 and the four development stages are described below.

- The properties of the interface structure of component x such as *functionality*, *static security knowledge base (KB)* and *execuatable code* are designed and implemented during the component development phase known as *construction stage*. *Static security KB* contains the ensured and required security properties related to the *functionality*. *Executable code* checks the compatibility of security properties for a composition.
- *Component ID* is provided by a certifying authority approving the exposed *static security KB* and *functionality* properties. It is done during the certification process in the component packaging phase.
- The CsC is a compositional security contract(CsC)

IEEE COMPUTER SOCIETY

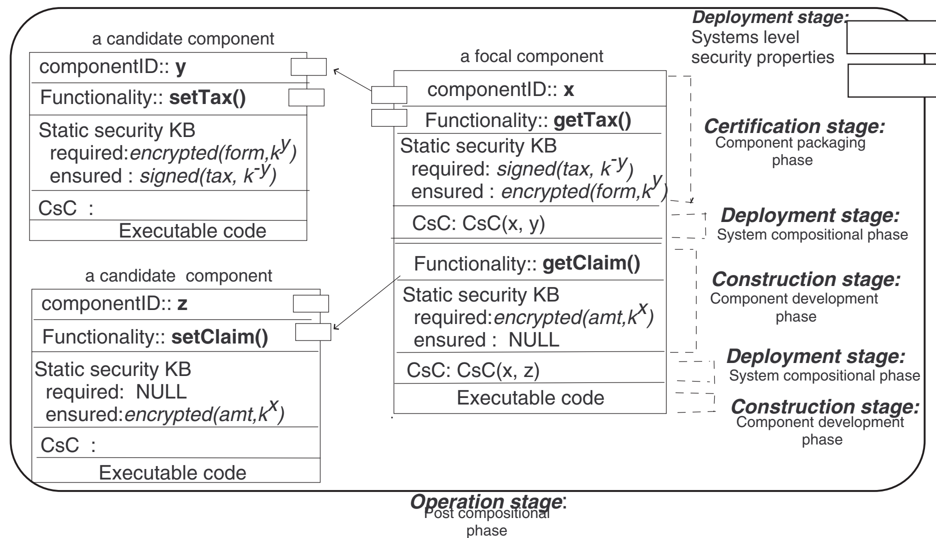

Figure 2. An example of the artefact model

between components which is reasoned about by the *executable code* as the component being assembled with other components in the compositional phase at the *deployment stage*. The established CsC is stored in the CsC slot.

- When the entire composed system is running and providing a meaningful service to the users, the components are said to be in the post compositional phase known as *operation stage*. The systems level security properties are characterised and registered in an independent register.

Our process framework advocates a pragmatic approach for implementing a self-configuring and composable component characterisation framework, relying mostly on the interface structure of atomic components. More on such interface structure could be found in [8, 9, 10]. In our framework, a component interface structure [10] not only contains the operations and attributes that the component provides to serve a functionality, but also embodies security properties associated with a particular functionality. The interface has the ability to 'publish' the component's own properties, and to 'read' and 'verify' the compositional security properties offered by other components. An interface of a component can access the security properties of another component or an existing composition, and compute the CsC of the dynamic system configuration and re-configuration. The artefact model supports a four-phase compositional model which is consistent with the processes we have outlined earlier.

- Firstly, an interface of a component publishes the static security properties attached with a functionality to its environment
- Secondly, the *executable code* tests for a possible CsC between the focal component and the candidate component
- Thirdly, if it succeeds, the results are stored in the *CsC* slot to be used later to configure and re-configure the composition.
- Finally, the security properties of the entire composite system based on the *CsC* are entered in a registry.

Once the security properties of the component functionality are specified at the *construction stage*, the component needs certification along with the interface structure at the *certification stage* before it is available for *deployment* and *operation*.

3. Tasks of the characterisation process

We now explain each of the four stages of the characterisation process framework in the following subsections.

3.1. Characterisation of atomic components

Figure 3 elaborates the first phase of our framework — the process of *characterising security properties of atomic components* cited in Figure 1. In brief, the following tasks in the construction stage are followed in order to characterise security properties of atomic components:

IEEE COMPUTER SOCIETY

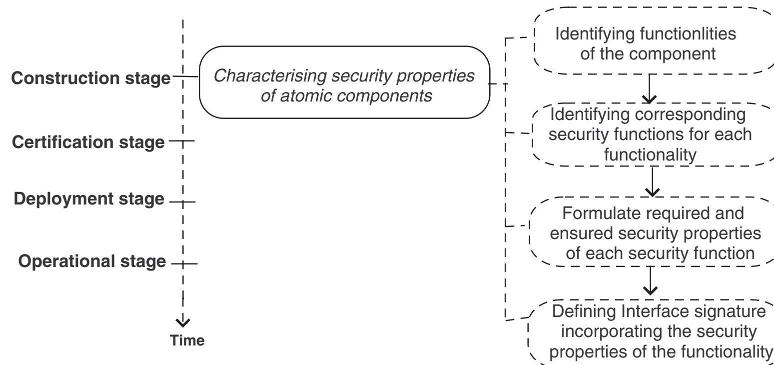

Figure 3. Phases in security characterisation process for atomic components

1. The first task identifies all functionalities that a component offers to its environment such as $f_1, \ldots, f_n$. This is a typical design level task in any component development process. The examples of such f_i could be $calculateTax$, $findMaxValue$, $printReport$ and so on. It is the functionality of the component in which other parties are most interested in. The main objective of identifying the 'functionality' is to establish the purpose or intention of a component in a compositional contract. In the example artefact model (see Figure 2), component x has two required functionalities i.e. *getTax()* and *getClaim()*, component y provides one functionality *setTax()*, and component z provides a functionality called *setClaim()*.

2. The second task identifies the security functions that are implemented with each of the functionality f_i. The security policies are usually implemented as security functions with the functionality. Each functionality may have its own security functions. Assume that component x has two security policies related to its functionality *getTax()*: (i) the object tax $form$ is encrypted whenever it is transmitted to another component; and (ii) it would not accept the tax object from other component unless it is digitally signed by the sender. The functionalities of the components y and z also have their own security policies.

3. The third task extracts the publishable security properties in terms of *required security properties* and *ensured security properties* of the functionality f_i from the identified security functions. In delivering a service, and assuming the required security properties are satisfied, the component may in return *ensure* certain security properties to the external world. In our example in Figure 2, the required and ensured properties of the functionality *getTax()* are:

   ensured: $encrypted(form, k^y)$.
   required: $signed(tax, k^{-y})$.

   The first property ensures that the object $form$ must be encrypted with the public key of y. The second property states that the functionality *getTax()* requires the object tax must be digitally signed by y. The security properties of *setTax()* is:

   ensured: $signed(tax, k^{-y})$.
   required: $encrypted(form, k^y)$.

   Similarly, functionalities *getClaim()* of x and *setClaim()* of y have their own security properties as depicted in Figure 2. Functionality *getClaim()* has only required property and *setClaim()* has only ensured property. These are self-explanatory.

4. The final task publishes the extracted *required* and *ensured* security properties to the environment in machine and human readable form. This phase addresses the issue of externalisation of the identified security properties to the environment. In our artefact model, the interface of component x exposes the publishable security properties to its environment through its *Static security KB* as shown in Figure 2. Similarly, other components y and z express their security properties through their *Static security KB* slots.

We have chosen a declarative notation which is based on logic programming to express security properties of components. More on these could be found in [9]. In principle, component developers could use a variety of languages to express security properties. The language could be a propositional or constraint language. For instance, the security properties could be written in XML for storage purposes.

1530-0803/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

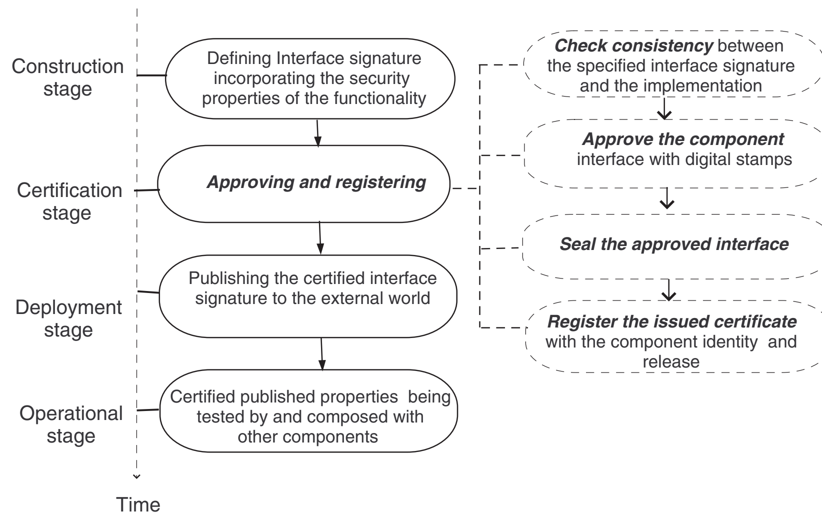

Figure 4. Certification Process in Security Characterisation Framework

3.2. Certification

The certification process ensures that the implementation of the component matches the published functionality and the security properties exposed through the component interface. All these need to be verified, certified, digitally stamped [4], and sealed by a certifying authority. A certified interface can further reveal more identity related information if queried such as details about the certificate, certification stamp, validity period, and so on. All identity and certification related information is read-only and public. Only the issuing certifying authority could alter those.

Figure 4 shows various stages (see solid line rounded rectangles) of interface development life cycle related to the designing, approving, publishing and checking of interfaces. The figure also illustrates the process of certification called *approving and registering* shown with the rounded dotted rectangles. The process has four tasks: (i) *check consistency*; (ii) *approve the component interface*; (iii) *seal the approved interface*; and (iv) *register the issued certificate of the component*. The certifying authority issues an unique component identity which is readily accessible from the interface of the component to others. Software components can only be tested and certified individually –not within the context of the complete composed system [6]. More on component certification are available from [1, 4, 14]. The certified assurances must be verifiable statically and dynamically by other components or the software integrators. Once certified, a re-compilation of the component would automatically erase all certification and identity related information stored in the template. In fact, a tampered interface would result in a void certificate, and this could be established by the contacting components from the information stored in the interface signature of the component. If the component needs to alter its security properties, it requires a new certificate after the re-compilation.

The security properties of component could be based on the specifications provided in the *functional and assurance requirements* defined in the *Common Criteria (CC) for Information Technology Security Evaluation, ISO/IEC 15408* [13]. If we characterise the security properties defined in CC, and use them in the component interface with proper sealing technique using a digital signature of a CA, it would significantly increase the user confidence on software components.

3.3. Compositional security contract (CsC)

In any characterisation framework, a component's fundamental capability to 'test' the composability of security properties of other component is most desirable. Not giving this capability of deciding the security contracts to the individual component would be denying the very nature of autonomous characteristics of software components. A CsC is based on the compatibility between the required security properties of a component and the ensured security properties of another component [10].

The establishment of a compositional contract is based on some fundamental elements:

- Universally unique identities of focal and candidate components

IEEE COMPUTER SOCIETY

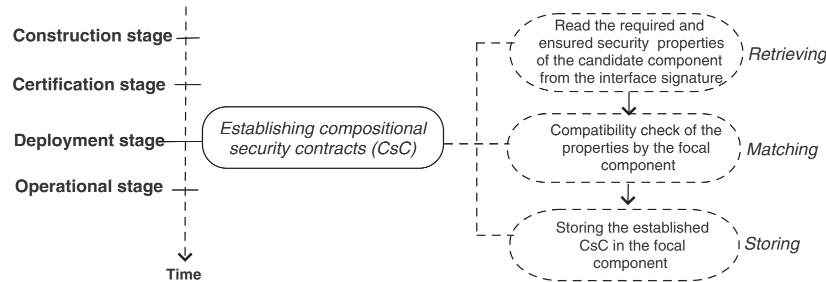

Figure 5. Process of Compositional Security Contract

- Unique identities of the requested functionality within a component
- Machine readable description of security properties,
- Algorithms to 'test' the composability of published security properties, and
- Storing the established CsC at the component level.

Figure 5 depicts the process of establishing a compositional security contract (CsC). The process of establishing a CsC consists of three tasks as shown with rounded rectangles: $retrieving$, $matching$, and $storing$.

1. In the $retrieving$ task, the required and ensured properties of a candidate component are accessed and read by the focal component. These properties are specified during the development of the atomic components as discussed in the previous section. In our artefact model cited in Figure 2, component x could read the security properties exposed in the static security KB of components y and z.

2. In the $matching$ task, the security properties retrieved in the previous task are checked against the security properties of the focal component. If all matchings are evaluated to *True*, then the next task processes the result of the matching. If no match is found, CsC may not be established. If a match is found, the focal component completes the checking. In our example, the $executable\,code$ of x reasons about the security properties of y against the security properties of x in relation to the functionality *getTax()*. For *getClaim()*, security properties of component z are tested against the security properties related to *getClaim()* and *setClaim()*. The actual CsC rules could be quite complex. Examples of such rules for CsC between *getTax()* and *setTax()* could be:
   $signed(tax, k^{-y}) \longleftarrow encrypt(form, k^{y})$.
   $CsC(x, y) \longleftarrow signed(tax, k^{-y})$.

   Similarly, rules for compositional contract between *getClaim()* and *setClaim()* could be:
   $CsC(x, z) \longleftarrow encrypted(amt, k^{x})$.

   In the above rule expression, the left hand side of the $\longleftarrow$ is the ensured property and the right hand side properties are required properties. The first rule specified for the CsC between x and y states that the object tax is digitally signed with the private key of y if the object $form$ is encrypted with the public key of y. The second rule states that a CsC between x and y is established if tax is digitally signed by y. The rule specified for the CsC between x and z states that there is a requirement from the component x that the amount amt is to be encrypted by z with the public key of x. More on this could be found in [9].

3. In the $storing$ task, the established compositional security contract (CsC) is stored in the CsC slot of the focal component. The resulting security contract of the composition participated by two components is the assurance of that particular composition [10]. In our example, component x has established two CsCs –one for each of its two functionalities.

The number of components involved in a composition determines the complexity. In dynamic run time composition, the functionality may be highly distributed among components residing remotely. The compositional effort of components may be reduced due to limited communication channels and minimal computing resources. It is therefore important to use observations or security reflection instead of exchanging increasingly detailed information to test and verify all security properties of components [3].

3.4. System-level security contracts (SsC)

A system-level security contract (SsC) of a composite system is established based on CsCs. Several CsCs con-

1530-0803/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

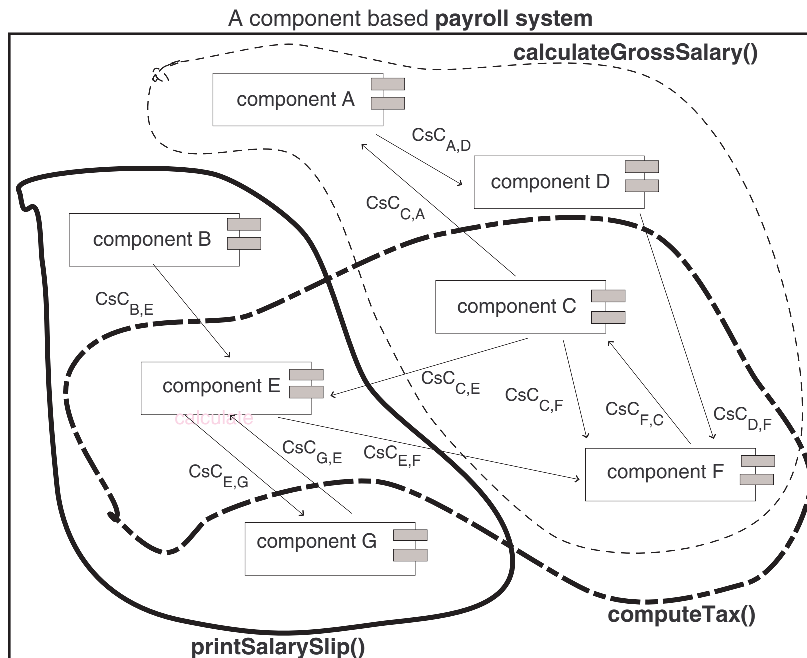

Figure 6. A systems-level security contract

tribute to make a system-wide security characterisation. A SsC simply exposes security properties of the composite system that are visible to the external world such as end-users of the application. In fact, a global federated system can be viewed as a coarse grained component. The security characterisation of the next level 'composite' system can also be specified in terms of CsCs established among various components to achieve one or more functionalities. Systems-level characterisation is very much dependent on the functionalities that are provided by the federated composite system to the environment.

Figure 6 shows an example of a component based payroll system. The figure shows a system composed of seven components creating three different functionalities. For instance, the solid line boundary encircling components B, E, and G constitutes a functionality called $printSalarySlip()$. This functionality has three CsCs namely, $CsC_{B,E}$, $CsC_{G,E}$, and $CsC_{E,G}$. The components C, E, and F within the thick broken line boundary constitute another functionality called $computeTax()$. This functionality consists of $CsC_{E,F}$, $CsC_{C,E}$, $CsC_{C,F}$, and $CsC_{F,C}$. Components A, C, D, and F constitute a functionality called $calculateGrossSalary()$ as shown with a rather thinner dotted boundary. The security properties of this functionality are based on $CsC_{A,D}$, $CsC_{C,A}$, $CsC_{C,F}$, $CsC_{F,C}$, and $CsC_{D,F}$. The security properties of a SsC are those that are attached with each of the functionality. However, the security properties of the global system are attached with the entire system, in this case, it is a payroll system. A registry stores the global security properties of the system *payroll system* in this case. The entire payroll system can also be treated as an atomic component if it needs to be composed as a whole with another system or component.

A system can be configured and reconfigured incrementally as components are added and detached. Hence, the configuration of the composite application changes ac-

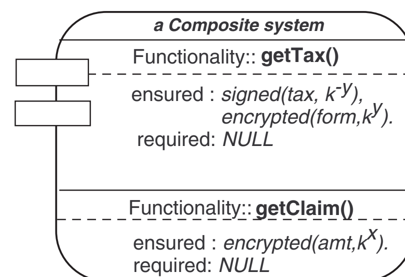

Figure 7. A systems-level security contract

IEEE COMPUTER SOCIETY

cordingly. In a composite system, security properties of one component may be consumed off by other participating components. However, a particular type of security property at the component level may be considered quite differently at the system level. For instance, a security property protecting an external communication channel between two components at the component level could be treated as protecting internal communication at the systems level –at least logically.

In our example artefact model cited in Figure 2, the federated system composed of three components, namely x, y and z could be used as an independent component by its environment. Figure 7 shows the ultimate security properties of the composite system worked out from the established CsCs among three components x, y, and z cited in Figure 2. Note that the SsC of the system is quite different than the security properties of the component x. Figure 7 shows the registry entry of the SsC of the composite system. A simple version of the algorithms used to form the SsC is shown in Figure 8. A brief description of each line in Figure 8 is given below.

(1) $f = determine(allFunctionality, G);$
(2) $CsC_{1,\ldots,j} = identify(all_C sC, f);$
(3) $init = identifyInit(CsC_{1,\ldots,j});$
(4) $R_{init} = findR(init);$
(5) $R_{s_{1,\ldots,m}} = findR_All(CsC_{1,\ldots,j});$
(6) $FOR\ k = 1 \rightarrow m\ DO$
(7) $\quad SsCR_k = NOTmatched(R_k, E_{CsC_{1,\ldots,j}});$
(8) $E_{s_{1,\ldots,m}} = findE_All(CsC_{1,\ldots,j});$
(9) $SsC_f = list(R_{init}, E_{s_{1,\ldots,m}}, SsCR_{1,\ldots,k});$

Figure 8. Algorithms to compute SsC

1. Identify the functionality f of the composite system G in which the user is currently interested in.
2. Identify all participating $CsC_{1,\ldots,j}$ involved in a particular functionality f of the composite system G.
3. Identify the component $init$ (from the CsCs) which acts as an initiator or a trigger of a functionality f that the composite system G offers. Such an initiator acts as the first interface of the composite system. $init$ is one of the components identified in $CsC_{1,\ldots,j}$.
4. Identify the required security properties R_{init} of the initiator component $init$ which need to be satisfied by entities external to the composite system G in order to initiate the processing of the functionality f. Only these required properties of $init$ need to be identified as well as satisfied by the external entities.
5. Identify all required security properties Rs of all participating components in $CsC_{1,\ldots,j}$.
6. Identify required properties $SsCR_{1,\ldots,k}$ from $R_{s_{1,\ldots,j}}$ which need to be satisfied by entities other than the participating components in the composite system G in order to accomplish the functionality f. Note that some other additional systems level security properties may be included in SsC. Even some properties at the component level may take a different form at this level.
7. Identify all ensured properties of the participating components as $E_{s_{1,\ldots,j}}$. Again, more additional systems level properties may be added into the SsC.
8. Finally, list the identified required property R_{init} of the initiator $init$, all identified required properties $SsCR_{1,\ldots,k}$ and $E_{s_{1,\ldots,j}}$ as the required and ensured security properties respectively of the functionality f of the composite system G.

4. Related work

In this section we give a brief citation of major work related to ours. Interface description languages (IDLs) deal basically with the syntactic structure of the interface and some semantics such as the forms and types of the interface elements like attributes, operations and events. These metadata are primarily static in nature. IDLs allow a contract to be defined between the service provider and the client [12]. For example, interfaces used in CORBA only contain the functional description. It does not allow the selection of objects on the basis of non-functional properties of the components. Although some of the existing approaches support management of non-functional properties but they still do not support the automatic runtime assessment of non-functional properties. Recent proposal for a richer interface specification addresses the issues of software component interface signature (syntax), interface configurations (structure), interface behaviour (semantics), and interaction protocol (constraints) [5]. Some recent developments in the standarisation of interface definition aim to simplify system composition such as XML, its recent extension ebXML [11], and a language called Xelha [2]. These focus on the issues of multimedia quality properties, data formats and interoperability, but do not address the integration of non-functional properties [12].

5. Conclusion

The proposed characterisation process framework lets component developers define component's functions-specific security properties, essentially providing security

IEEE COMPUTER SOCIETY

oriented ontologies based on taxonomies of terms, relationships and compositional rules. Other components as well as software integrators can understand and manipulate those published ontologies to discover and assess the properties for a viable composition. The framework allows the participating components to reveal their security properties, identity and origin to each other. The framework further allows the participants to 'check' upfront the consistency of those properties for a viable composition. The principal objective is to generate computational reflection to let components identify and capture various security properties of other components with which they cooperate.

Our process framework represents security properties of a component in a declarative form instead of codifying them in the application source code. With this approach, software composers can predict how the overall security properties of the composed system look like. When security properties are spelled out in simple comprehensible terms, systems integrators are then better informed about what to expect from and what to provide to components. The characterised security properties exposed with the interface can assist the integrators to decide whether a component would be able to meet the specific security requirements of their application or not. A well defined security characterisation that clarifies the security responsibility of the component can inspire trust.

References

[1] W. Councill. Third-party testing and the quality of software components. *IEEE Software*, pages 55–57, July-August 1999.

[2] A. Duran-Limon and G. Blair. Specifying real-time behaviour in distributed software architectures. In *Proceedings of the Third Australasian Workshop on Software Systems Architectures*. Monash University, 2000.

[3] E. Durfee. Scaling up agent coordination strategies. *IEEE Computer*, 34(7):39–46, July 2001.

[4] A. Ghosh and G. McGraw. An approach for certifying security in software components. In *Proceedings of the 21st National Information Systems Security Conference*, 1998.

[5] J. Han. A comprehensive interface definition framework for software components. *IEEE Proceedings Asia Pacific Software Engineering Conference*, pages 110–117, 1998.

[6] J. Hopkins. Component primer. *Communications of ACM*, 43(10):27–30, October 2000.

[7] R. Keller, B. Lague, and R. Schauer. Report on the international workshop on large-scale software composition. *ACM Software Engineering Notes*, 24(1):49–54, January 1999.

[8] K. Khan and J. Han. Security aware software composition. *IEEE Software*, 19(1):34–41, Januray/February 2002.

[9] K. Khan and J. Han. A security characterisation framework for trustworthy component based software systems. In *Proceedings of the COMPSAC*, pages 164–169. IEEE Computer Society, 2003.

[10] K. Khan, J. Han, and Y. Zheng. A framework for an active interface to characterise compositional security contracts of software components. *IEEE Proceedings Australian Software Engineering Conference*, pages 117–126, 2001.

[11] D. Linthicum. *B2B Application Integration*. Addison-Wesley, 2000.

[12] G. Ribeiro-Justo and A. Saleh. Non-functioanl integration and coordination of distributed component services. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2002.

[13] Standard. Common criteria for information technology security evaluation, ISO/IEC 15408. Technical Report v2.0, Nat'l Institute Sandards and Technology, Washington, 1999.

[14] J. Voas. Certifying software for high-assurance environments. *IEEE Software*, pages 48–54, July-August 1999.

1530-0803/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY