Composing Security-Aware Software

Khaled M. Khan, *University of Western Sydney*

Jun Han, *Monash University*

The resurgence of component-based software development offers software engineers new opportunities to acquire third-party components to deliver system functionality. Component-based software engineering represents the concepts of assembly and coupling of components—essential to most engineering disciplines. The development paradigm of coupling and decoupling software components is receiving a great deal of interest from industry and academia, as it promises maximum benefits of software reusability. While software components have become popular, security concerns are paramount. Their composition can be considered risky because of the "plug and play" with unknown third-party components. In dynamic runtime applications for critical systems such as e-commerce and e-health, the risk could be much higher.

Component security concerns are twofold: how to build secure components and secure composite systems from components, and how to disclose components' security properties to others. This article addresses the latter; rather than propose any new security architecture, we present a security characterization framework. Our approach concerns the security functions of software components by exposing their required and ensured security properties. Through a *compositional security contract* between participating components, system integrators can reason about the security effect of one component on another. A CSC is based on the degree of conformity between the required security properties of one component and the ensured security properties of another. However, whether the characterized and disclosed security properties suffice to build a secure composite system is outside the scope of this parameter. System integrators should address this concern at the time of composition.

This article introduces a component security characterization framework that exposes security profiles of software components to inspire trust among software engineers.

Component mistrust

There is now an open challenge on how to cultivate and inspire software developers' much-needed trust in third-party components. The attributes that most affect a trust relationship are identity, origin, and security properties that components offer to and require from other components. If the developer doesn't know these attributes during system integration, the component might not

 0740-7459/02/$17.00 © 2002 IEEE

be trustworthy. In current practice, the trust-related attributes are often neither expressed nor communicated. Software developers are reluctant to trust a third-party software component that does not tell much about its security profile. Despite these shortcomings, software engineers are still inclined to use them to minimize development effort and time. Today, trust in an application system is based on consent—that is, the user is explicitly asked to consent or decline to use a system.[2] At the application level, such consent-based trust perhaps works fine. But in a component-based development environment, universally shallow commitment regarding component security is dangerously illusive and can trigger costly consequences.

Trust requirements in a development environment significantly differ from those of application users. Component security—based on various nondeterministic elements such as the use domain, magnitude of the hostility in the use context, value of the data, and other related factors—is relative, particularly in a component-based development environment. Therefore, software engineers must be assured with more than just a component security or insecurity claim. Whatever small role a component plays, the software engineer cannot rule out its possible security threats to the entire application.

Component developers might not be aware of the security requirements of their products' potential operational contexts. Software engineers do not expect such knowledge from the component developer, but they do expect a clear specification of the component security requirements and assurances.[1] This information should be made available if queried at runtime. Developers must be able to do runtime tests with candidate components to find possible security matches and mismatches.

The major concern—the disclosure of components' security properties and security mismatches of those properties—has received little attention from the security and software engineering research communities. Current practices and research for security of component-based software consists of several defensive lines such as firewalls, trusted operating systems, security wrappers, secure servers, and so on. Some significant work on component testing, component assurances, and security certification has been done, particularly in the last two years.[3-5] These efforts basically concentrated on how to make a component secure, how to assure security using digital certification, and how to maximize testing efforts to increase the quality of individual components. Undoubtedly, such work is important to inspire trust, but we must explore other possibilities that would let software engineers know and evaluate the actual security properties of a component for specific applications.

There is now an open challenge on how to cultivate and inspire software developers' much-needed trust in third-party components.

Our approach

Driven by these ideas and motivations, we propose a security characterization framework in this article. The framework addresses how to characterize the security properties of components, how to analyze at runtime the internal security properties of a system comprising several atomic components, how to characterize the entire system's security properties, and how to make these characterized properties available at runtime. To inspire trust in a particular composite system, a component's security contract with all the other components, the security provisions that each component requires from ensures to the others, and the ultimate global security profile of the entire federated system should be clear.

Atomic components

Security properties and behaviors of a software system are categorized into 11 classes in ISO/IEC-15408 Common Criteria.[6] These classes are made of members, called families, based on a set of security requirements. We will only discuss a subset of one such security class, *user data protection*, just to give a snapshot of our characterization framework.

The publishable security properties related to user data protection of any atomic component can be categorized as *required*—a precondition that other interested parties must satisfy during development to access the ensured security services—or *ensured*—a postcondition that guarantees the security services once the precondition is met. Security properties are typically derived from security functions—the implementation of security policies. And the security policies are defined to withstand security threats and risks. A simple security function consists of one or more principals (a principal can be a

The final and important issue in a security characterization framework is how to make components' security profiles and CSCs available to other components.

human, a component, or another application system, whoever uses the component), a resource such as data, security attributes such as keys or passwords, and security operations such as encryption. Based on these, three main elements characterize an ensured or required security property: *security operations* executed by the components to enforce security properties, *security attributes* required to perform the operation, and *application data* manipulated in a compositional contract.[7] Using these elements, we can formulate a simple structure to characterize the security requirements and assurances of individual components:

$$f(\mathrm{O}_i, \mathrm{K}_j, \mathrm{D}_k)$$

where *f* represents a *security objective* formed with three associated arguments; O is the security-related *operation* performed by the principal *i* in a compositional contract; K is a set of *security attributes* used by the principal; subscript *j* contains additional information about K such as key type, the key's owner, and so on; D is an arbitrary set of data or information that is affected by the operation O; and the subscript *k* contains additional information regarding D such as whether a digital signature is used or not. The following examples represent a required security property *R* (protect_in_data) and an ensured security property *E* (protect_out_data) of a component *P*:

$$R_P = \text{protect_in_data (encrypt}_Q\text{, key}_{P+}\text{, 'amount')}$$
$$E_P = \text{protect_out_data (encrypt}_P\text{, key}_{Q+}\text{, file1}_{\text{P}-\text{.digi_sign}}\text{)}.$$

In this example, component *P*'s required property R_P states that the data is to be encrypted by any component *Q* with component P's public key. A plus sign (+) after *P* denotes public key. The ensured property E_P states that component *P* encrypts the data file with the public key of any component *Q*. The data is also digitally signed by *P* with its private key, denoted by the minus sign (–) after *P*. This format is specific to a particular type of security function related to user data protection. This notation, or a similar one, can be standardized for all components. However, alternative structure might need to be formulated to represent other security classes such as authentication, security audit, trusted path, privacy, and so on.

Analysis of security contracts

A component that broadcasts an event to receive a service is called a *focal component*. Software components that respond to the event are usually called *candidate components,* and they might reside at different remote locations. With the security characterization structure of atomic components previously explained, a CSC between two components such as *x* and *y* can be modeled as

$$C_{x,y} = (E_y \Rightarrow R_x) \wedge (E_x \Rightarrow R_y)$$

where *C* is a compositional security contract between focal component *x* and candidate component *y*. The expression $E \Rightarrow R$ denotes that *E* implies *R*, or *E* satisfies *R*. The *required* or *ensured* security property of an existing CSC can be referred to as $C_{x,y}.R_y$ or $C_{x,y}.E_x$ respectively. The degree of conformity between the required security properties of one component and the ensured security properties of another is the ultimate CSC of the composite system.

Global security characterization

As is the case of atomic components, we also need to establish a global security characterization of a composite system, because it might be used in further composition as a component. In fact, developers often view this kind of system as a single entity or an atomic component, not as a collection of components in such further components. Therefore, the CSC approach can specify the composite component security by required and ensured properties based on the structure and principles defined for the atomic component. However, such a characterization depends on the functionality that the composite system provides. We can derive the externally visible security properties—that is, the required and ensured properties—of a composite component based on CSCs among participating components.

Notion of an active interface

The final and important issue in a security characterization framework is how to make components' security profiles and

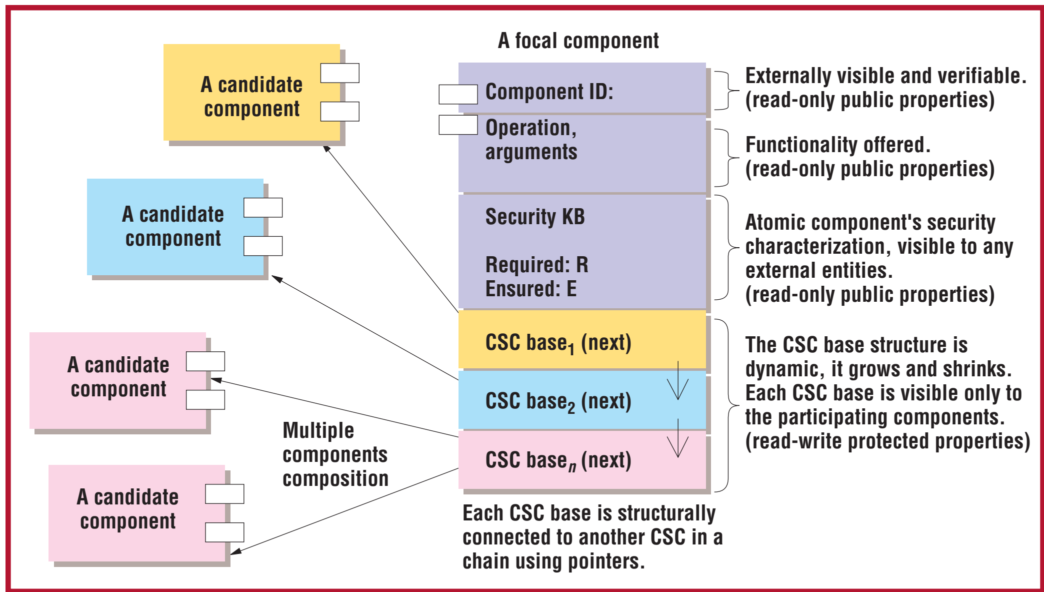


Figure 1. Structure of an active interface.

CSCs available to other components. This article extends the CSC model to make it active, in the sense that each interface between components will have certain reasoning capability. We call them *active interfaces*. Current frameworks for software component models such as EJB, Corba, COM, and .Net are limited to the specification and matching of structural interface definitions.[8] Interface description languages (IDLs) deal with the syntactic structure of the interface such as attributes, operations, and events. In our approach, an active interface not only contains the operations and attributes to serve a function but also embodies the security properties associated with a particular operation or functionality. An active interface supports a three-phase automatic negotiation model for component composition:

- A component publishes its security properties attached with a functionality to the external world.
- The component negotiates for a possible CSC at runtime with other interested candidate components.
- If it succeeds, the negotiation results are used to configure and reconfigure the composition dynamically.

Active interface structure

An active interface consists of a *component identity*, a static *interface signature*, a static (read-only) *security knowledge base* of the component, and a (read–write) *CSC base* that is dynamic based on the information available from the security knowledge base.[7] Before a component is available for use, a certifying authority must certify it. A certificate ensures that the implementation matches the published functionality and the exposed security properties. It is argued that software components can only be tested and certified individually—not within the context of the complete composite system.[8] The certified assurances must be verifiable statically and dynamically. Figure 1 illustrates a skeleton of an active interface structure.

The ComponentID in the active interface includes a unique identity (UID) provided by a certifying authority, the component's current residing address (URL), details about the component developer, and the certification authority that certified the component:

ComponentID (uid, URL,
developer_ID, certificate)

A certifying authority will verify, certify, and digitally stamp all of this data.[5] It can further reveal more identity information if queried about the certificate, certification stamp, validity period, and so on. All identity and certification information is read-only and public—only the certifying authority can alter it.

Operations and arguments

An interface signature consists of operations and attributes for a particular functionality. These operations and attributes

If the component needs to alter its security properties, it requires a new certificate after the recompilation.

are used for structural plug-and-play matching. These properties are static—read-only properties. Components cannot make any modification to this. This interface is intended to make a structural match before two components are composed.

Security knowledge base

A security knowledge base stores and makes available the security properties of a component in terms of $f(O_i, K_j, D_k)$. The required and ensured properties stored in this KB are specific to the functionality that the component offers. These properties must be based on the actual security functions that the component uses to accomplish a particular functionality. A component might offer various functions, so the exposed security properties can vary accordingly.

Once the information is stored in a KB and certified, no other entities can alter its content. Any recompilation of the certified component would automatically erase all certification and identity information stored in ComponentID. If the component needs to alter its security properties, it requires a new certificate after the recompilation.

CSC base

A binary executable piece of code residing in the active interface of the focal component generates CSC conformity results between the focal component and a candidate component. If the system identifies nonconformance between the required and ensured properties it concludes with a security mismatch. The resulting CSC is automatically stored in the CSC base of the focal component, and remains there as long as the composition is valid.[7] Also, a component can accept a partially or completely mismatched CSC, although this might have negative security effects on the global system. If a component becomes obsolete or is no longer needed in a dynamic composition, the associated obsolete CSC might be stored in a log belonging to the focal component for future audit purposes, but it would not be available to any of the participating components.

An example

We use a fictitious distributed-system topology as an example of how our proposed active interface would work in a distributed environment. Consider an e-health care system that regards all clinical information passing among the stakeholders, such as the general practitioners, specialists, patients, and pharmacists, as confidential. Assume a focal component *Y* running on a machine at a GP's office connects with a trusted candidate component *S* chosen from among many such systems running at various specialists' offices. *Y* provides a patient's diagnosis report to *S* to get a prescription. After receiving the prescription from *S*, *Y* sends it electronically to a candidate component *P* residing on a pharmacist's system for a price quotation. Developers would independently develop many such *P*s and *S*s and make them available from their various distributed sources, potentially able to deliver the functionality that *Y* wants.

However, component *Y* not only is interested in specific functionality but also wants to know upfront the security properties that those components provide.

Binary CSC

Assume component *Y* exposes the following required and ensured security properties:

```
SECURITY {
 REQUIRED_Y {R_Y =
   protect_in_data (encrypt_S,
   key_S-, 'prescription'_S.digi_sign)}
 ENSURED_Y {E_Y = protect_out_data
   (encrypt_Y, key_S+, 'diagnosis')}}.
```

The ensured property states that *Y* will provide a diagnosis report of a patient to a specialist component. *Y* would encrypt ($encrypt_Y$) the report with *S*'s public key (key_{S+}). In return, *Y* requires from *S* that *S* digitally sign ($'prescription'_{S.digi_sign}$) and encrypt ($encrypt_S$) the prescription it sends with its own private key (key_{S-}).

Now assume that in response to the event *Y* broadcasts for the functionality Get_prescription, it receives responses from components *S*1 and *S*2 offering that functionality. *S*1 and *S*2 run on different machines for different specialists; they are independently developed and serviced by different developers and have their own security requirements and assurances. *Y* also reads the certification information, origin, and identity of the components from the interfaces of *S*1 and *S*2. *Y* first queries *S*1. *S*1's interface exposes its security properties

stored in its static KB as

```
SECURITY {
 REQUIRED_s1 {R_s1 =
   protect_in_data (encrypt_Y,
   key_s1+, 'diagnosis')}
 ENSURED_s1 {E_s1 =
   protect_out_data (encrypt_s1,
   key_Y+, 'prescription')}}.
```

According to these security properties, component $S1$ requires that component Y encrypt the diagnosis report with $S1$'s public key. In return, $S1$ would encrypt the prescription with Y's public key, but $S1$ would not digitally sign the prescription data. Y's active interface now generates the CSC between Y and $S1$ based on

$$C_{Y,S} = ((E_Y \Rightarrow R_{S1}) \wedge (E_{S1} \Rightarrow R_Y)).$$

In the generated CSC, $S1$'s ensured security property has not fully satisfied Y's required security property, because $S1$ does not provide the digital signature with the prescription, as Y requires. After making a similar query to $S2$, Y reads $S2$'s disclosed security properties as

```
SECURITY {
 REQUIRED_S2 {R_S2 =
   protect_in_data (encrypt_Y,
   key_S2+, 'diagnosis')}
 ENSURED_S2 {E_S2 =
   protect_out_data (encrypt_S2,
   key_S2-,'prescription'_S2.digi_sign)}}.
```

Component $S2$ requires Y to encrypt the diagnosis report with $S2$'s public key. In return, $S2$ ensures that it would digitally sign (`'prescription'_S2.digi_sign`) and encrypt the prescription with its private key. Y can decrypt the message using $S2$'s public key to verify the signature. Based on these security properties, the generated CSC is consistent with the requirements of Y and $S2$. Y can finally be combined with $S2$. The resulting properties are now stored in Y's CSC base$_1$ for future reference.

Transitive CSC

We extend the same scenario further to examine how our framework can support a *transitive composition* with multiple components. A transitive CSC occurs when two components are composed in an extended sense—that is, when a CSC between two components is influenced by the ensured or required property of a third component or even by another existing CSC. After Y has combined with $S2$, it then looks for a third component P that would provide a price quotation for the prescription produced by $S2$. Y's security properties for this particular functionality are

```
SECURITY {
 REQUIRED_Y {R_Y =
   protect_in_data (encrypt_P,
   key_P-, 'price'_P.digi_sign)}
 ENSURED_Y{E_Y = protect_out_data
   (encrypt_Y, key_P+,
   C_Y, S2.E_S2.(prescription)_S2.digi_sign)}}.
```

According to these properties, Y agrees to attach $S2$'s digital signature (`C_Y,S2.E_S2.(prescription)s2.digi_sign`) to a component P to ensure that a specialist authenticates the prescription. In return, Y requires that P digitally sign and encrypt the price data. Note that these security properties of Y are quite different from those for the specialist prescription.

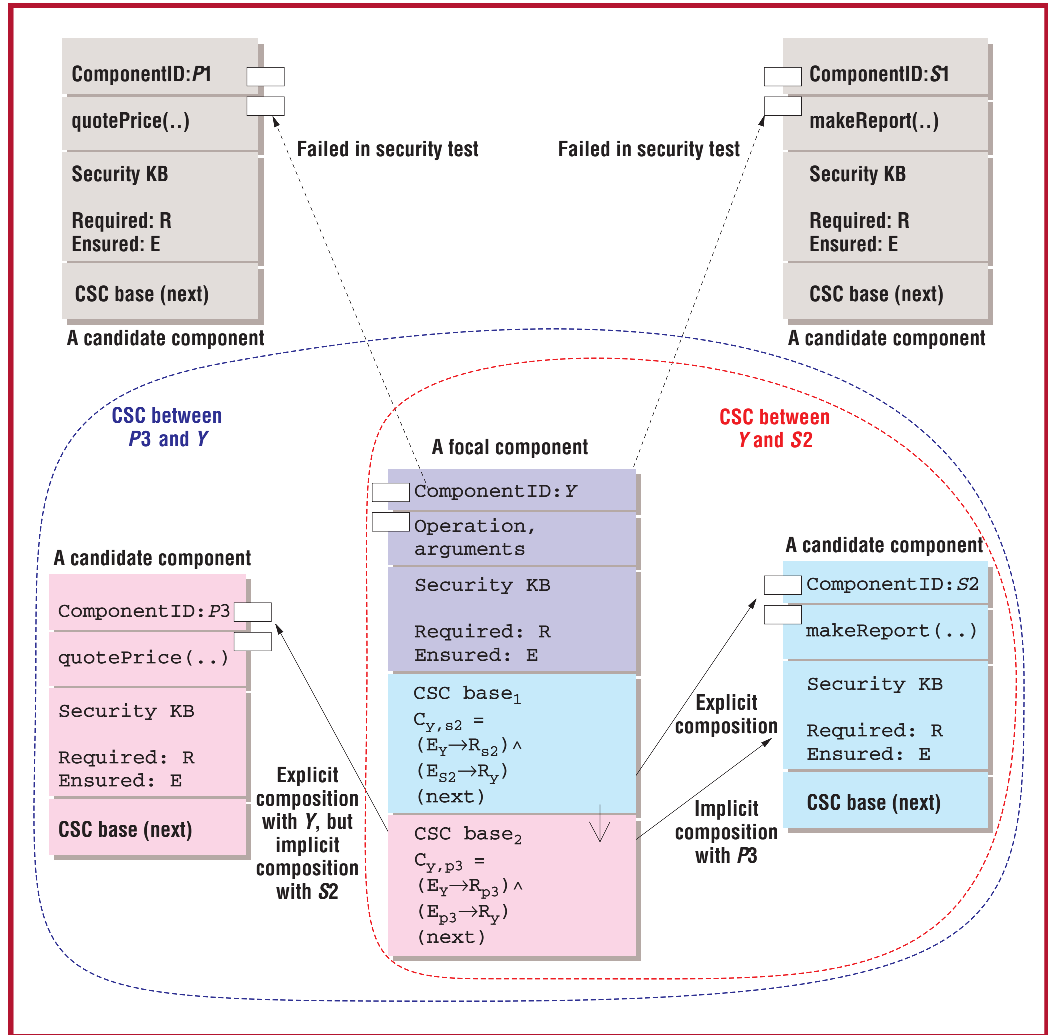
Now assume that in response to Y's broadcasting a request for a price quotation, remote components $P1$ and $P3$ have registered their interests in providing the functionality that Y wants. $P1$ and $P3$ are developed and serviced by two different development organizations and have their own security requirements and assurances. Y now runs a security test with $P1$ to verify whether the component could deliver the functionality as well as the security that Y requires. It also verifies whether Y by itself could satisfy $P1$'s required property. Y reads the security properties exposed by $P1$ as

```
SECURITY {
 REQUIRED_P1 {R_P1 =
   protect_in_data (encrypt_Y,
   key_Y-, (C_Y,S.E_S)_Y.digi_sign)}
 ENSURED_p1 {E_P1 =
   protect_out_data (encrypt_P1,
   key_P-, 'price'_P1.digi_sign)}}.
```

Y's interface generates a CSC. It shows that $P1$'s ensured security property has satisfied Y's required property, but that Y has not satisfied $P1$'s required property. This is because $P1$ requires a digital signature from

A component can accept a partially or completely mismatched CSC, although this might have negative security effects on the global system.

Figure 2. A composite system with two compositional contracts.



Y, but *Y* does not have one. Thus, *Y*'s security test with *P*1 has failed. *Y* now makes another security query to *P*3. *Y* reads the exposed security properties of *P*3 as

```
SECURITY {
  REQUIRED_P3 {R_P3 =
    protect_in_data (encrypt_Y,
    key_P3+, C_y,s.ES(prescription)
     s.digi_sign)}
  ENSURED_P3 {E_P3 =
    protect_out_data (encrypt_P3,
    key_P3-, 'price'_P3.digi_sign)}}.
```

Y's interface now generates the CSC with *P*3 based on

$$C_{Y,P3} = ((E_Y \Rightarrow R_{P3}) \wedge (E_{P3} \Rightarrow R_Y)).$$

The derived CSC is consistent with the required compositional contract between *Y* and *P*3. Satisfied with this CSC, *Y* combines with *P*3 and stores the CSC in its CSC base$_2$. Interestingly enough, the compositional contract $C_{Y,P3}$ involves three components in the relationship chain:

$$C_{Y,P3} = (((C_{Y,\ S2}.E_{S2((prescription)S2.digi_sign)\ encyrpt)} \Rightarrow R_{P3}) \wedge (E_{P3} \Rightarrow R_Y)).$$

The entire system scenario is shown in Figure 2. There are two CSCs in this system: one between *Y* and *S*2 (shown by the red dotted line) and the other between *P*3 and *Y* (shown by the larger blue dotted line).

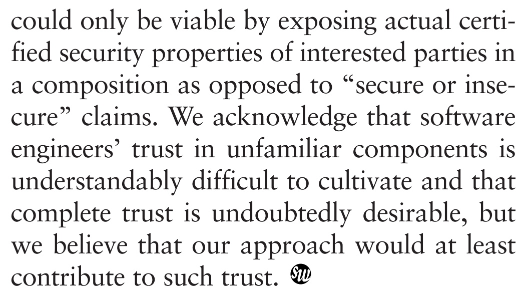
In the latter composition, *S*2 is *transitively composed* with *P*3 because *P*3's security requirements partly depend on *S*2's security assurances, although *P*3 does not have any direct composition with *S*2.

With the previous examples, we have demonstrated that software components

can know and reason about the actual security requirements and assurances of others before an actual composition takes place. The example also suggests that a security characterization is a mechanism to provide "informed consent."[2] An informed consent gives the participating entities explicit opportunity to consent or decline to use components after assessing the candidate components' security properties.

Our framework's main objective is to generate computational reflection to let components and their developers identify and capture the various security properties of the other components with which they cooperate. In such a setting, components not only read the metadescription of others' security properties but also identify security mismatches between two components and evaluate composability realistically. Security characterization and third-party certification of components would mutually benefit each other: first, a security characterization would contribute significantly to the process of component security certification; second, certification would make the exposed security properties more creditable to software engineers. When required and ensured security properties are spelled out in simple, comprehensible terms, software engineers are better positioned to evaluate the strength of the security a component provides. They are also well informed about what to expect from and provide to the component to establish a viable composition.

In a software engineering context, we must balance security against the other design goals of the entire component-based system. To achieve this, application developers must know about components' security properties. A trusting profile could be gradually built and inspired on the basis of the participating components' self-disclosure of their security properties. The security properties built into a component represent the efforts already put into place to withstand certain security threats. However, the real protection with the committed effort of the component from any security threat is beyond the control of the component. Whether the available resources disclosed by the component are sufficient to withstand a threat is outside the parameters of our framework. A trust-generating effort



could only be viable by exposing actual certified security properties of interested parties in a composition as opposed to "secure or insecure" claims. We acknowledge that software engineers' trust in unfamiliar components is understandably difficult to cultivate and that complete trust is undoubtedly desirable, but we believe that our approach would at least contribute to such trust.

Acknowledgments
The work reported here has benefited from earlier discussions with Yuliang Zheng while he was with Monash University.

References

1. D. Carney and F. Long, "What Do You Mean by COTS?," *IEEE Software*, vol. 18, no.2, Mar./Apr. 2001, pp. 82–86.
2. B. Friedman, P.H. Kahn Jr., and D.C. Howe, "Trust Online," *Comm. ACM*, vol. 43, no. 12, Dec. 2000, pp. 34–44.
3. J. Voas, "Certifying Software for High-Assurance Environments," *IEEE Software*, vol. 16, no. 4, July/Aug. 1999, pp. 48–54.
4. W. Councill, "Third-Party Testing and the Quality of Software Components," *IEEE Software*, vol. 16, no. 4, July/Aug. 1999, pp. 55–57.
5. A. Ghosh and G. McGraw, "An Approach for Certifying Security in Software Components," *Proc. 21st Nat'l Information Systems Security Conf.*, Nat'l Inst. Standards and Technology, Crystal city, Vir., 1998, pp. 82–86.
6. *ISO/IEC-15408 (1999)*, *Common Criteria for Information Technology Security Evaluation*, v2.0, Nat'l Inst. Standards and Technology, Washington, DC, June 1999, http://csrc.nist.gov/cc. (current Dec. 2001)
7. K. Khan, J. Han, and Y. Zheng, "A Framework for an Active Interface to Characterize Compositional Security Contracts of Software Components," *Proc. Australian Software Eng. Conf.*, IEEE CS Press, Los Alamitos, Calif., 2001, pp. 117–126.
8. J. Hopkins, "Component Primer," *Comm. ACM*, Oct. 2000, vol. 43-10, pp. 27–30.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.

Khaled M. Khan is a lecturer in the School of Computing and Information Technology at the University of Western Sydney, Australia. His research interests include software components, software maintenance, and software metrics. He received a BS and an Ms in computer science and informatics from the University of Trondheim, Norway, and another BS from the University of Dhaka, Bangladesh. He is a member of the IEEE Computer Society. Contact him at k.khan@uws.edu.au.

Jun Han is an associate professor in the School of Network Computing at Monash University, Australia, where he directs the Enterprise and Software Systems Engineering and Technology Laboratory. His research interests include component-based software systems, software engineering tools and methods, and enterprise systems engineering. He received a BEng and MEng in computer science and engineering from the Beijing University of Science and Technology and a PhD in computer science from the University of Queensland. He is a member of the IEEE Computer Society. Contact him at jhan@monash.edu.au.