The Saga Security System: A Security Architecture for Open Distributed Systems

Masakazu Soshi   Mamoru Maekawa

Graduate School of Information Systems,
University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, Tokyo, JAPAN 182

Abstract

In the paper we present an overview of Saga Security System*, a security architecture in open distributed systems. An agent in Saga Security System is called a* Saga Agent*.*

The authorization model in Saga Security System (Saga authorization model) *supports the novel concept of a* service path *and provides uniform and flexible protection appropriate for advanced computational models such as object-oriented systems and cooperative agent systems. With respect to the security mechanism of Saga Security System, its key features are an* access token *and a* Security Monitor*. Access tokens are implemented using public key technology and ensure integrity of request messages issued by Saga Agents. In addition, we can regard the Security Monitor of a Saga Agent as a* reference monitor *for the Agent. Security of a Saga Agent during its traversal over distributed environments is controlled by the Security Monitor integrated with the Agent.*

1 Introduction

In recent years, inexpensive and high performance computers come into wide use, and more and more computers become interconnected through computer networks. In such environments, we can enjoy easy sharing of resources over networks, although, we usually suffer from reduced security of such environments at the same time.

It is often the case that in conventional "secure" systems, these environments are not taken into account. However, such environments as described above are expected to dominate, and establishment of security of a proper level in them is of critical importance.

Motivated by these concerns, we are designing and developing *Saga Security System*[1], a security architecture in open distributed environments [9, 16, 17, 18]. In this paper we describe an overview of Saga Security System.

The remainder of the paper is structured as follows. We present the fundamental problem of security in open distributed systems in Section 2. To solve the problem, we propose Saga Security System and discuss it in Section 3. Next in Section 4, we explore Saga authorization model. We present the security mechanism of Saga Security System in Section 5. Then we review related work in Section 6. Finally we discuss the future work and conclude this paper in Section 7.

[1]We called this system "Security Agent System" before, but we have changed the name because it was too generic and a little confusing.

2 Security Problems in Open Distributed Systems

To begin with, let us consider the fundamental problem of security in open distributed systems.

In open distributed systems, among multiple nodes or applications, information is repeatedly copied or moved as it is. In such environments, once security is compromised in the middle of information flow and unauthorized dissemination of the information takes place, its security can never be controlled anymore. This is one of the most serious security problems inherent in open distributed systems.

Unfortunately, however, the use of mutual authentication protocols and encrypted communications for network security [13] alone is not a satisfactory solution to the problem, because once encrypted information is decrypted, it is not protected anymore. Hence, though the use of encryption technique provides communication security, the ultimate security of information depends on security of the computer that processes the information.

These considerations naturally lead us to the concept of mobile agents or objects [4, 7]: instead of information being delivered as it is (whether encrypted or not), a dedicated autonomous agent traverses over a network system. Such an agent is a self-contained entity from the viewpoint of security, i.e., it has a security profile for the information, security procedures, audit trail, and so forth. Additionally, information contained in an agent is accessed only through the agent, and the required level of security is attained.

This is the background of *Saga Security System*, a security architecture in open distributed environments. An agent in Saga Security System is called a *Saga Agent*.

3 Overview of Saga Security System

Saga Security System has the following outstanding features not found in traditional computer systems:

1. In order to apply to as many computing models and environments as possible, Saga Security System provides a general security architecture that is independent of programming languages and application semantics.

2. Under the condition of Item 1 above, Saga Security System gives *Saga authorization model*, which supports advanced computing models such as object-oriented systems and cooperative agent systems, in open distributed environments. For that purpose, Saga

authorization model defines the novel concept of a *service path* and provides flexible and powerful protection based on it.

3. A Saga Agent is a self-contained entity from the viewpoint of security. In particular, a *Security Monitor* is associated with every Saga Agent, and while the Agent travels over an open distributed system, its security can be controlled by the Security Monitor. Furthermore, *access tokens* are implemented using public key technology and ensure integrity of request messages issued by Saga Agents.

Therefore, Saga Security System realizes a promising security architecture in current open distributed environments.

Now we present an overview of Saga Security System. Saga Agents as mobile agents (objects) can be implemented in scripts, byte codes, or machine (binary) codes. As mentioned in Item 1 above, however, Saga Security System is supposed to provide a security architecture applicable to as many computing models and environments as possible. Therefore, we do not make any assumption about how a Saga Agent is executed[2]. Instead, we make some assumptions about a computational model of Saga Agents and about their execution environments, both of which are given below.

A *Saga Agent* is a self-contained entity from the viewpoint of security, where data and methods to access the data are encapsulated. In other words, a Saga Agent must be an instance of an abstract data type at the very least. Of course, it may or may not be completely object-oriented or intelligent. A method of a Saga Agent is called a *service*, and in the following discussions, $o.s$ stands for a service s implemented in a Saga Agent o. Finally, Saga Agents interact with one another via synchronous message passing.

Furthermore, a Saga Agent is executed on a *Saga Agent Base* (SAB). An SAB is an execution environment for Saga Agents and is supposed to be a Trusted Computing Base (TCB) [6]. An SAB guarantees that each Saga Agent runs in its own address space distinctly separated from others and that message passing is the only means when a Saga Agent wants to access other Agents. Moreover, an SAB supports secure migration of Saga Agents.

Under these assumptions, in the paper we discuss Saga Security System, focusing on its authorization model (Saga authorization model) and security mechanism. For more details of Saga Security Systems, see [9, 16, 17, 18].

4 Saga Authorization Model

In this section, we discuss Saga authorization model, which provides uniform and flexible protection appropriate for advanced computational models such as object-oriented systems and cooperative agent systems. The reader is referred to [18] for further details of the model.

4.1 Service Context

In traditional authorization models, users can exercise the privileges of directly invoking primitive operations, such as read and write, and these models cannot take advantage of the concept of encapsulation in object-oriented systems [1]. To counter the problem, Saga authorization model should specify not only authorizations to invoke primitive operations, but also authorizations to invoke services provided by Saga Agents. Additionally, in Saga authorization model, if we can grant authorization on services to a principal as well as *to another service*, we can provide flexible protection scheme [2].

Moreover, Saga authorization model must take current distributed environments into consideration. Therefore, the model should be able to control security and trust relationships of Saga Agents. For example, in distributed environments, it is desirable for the model to be able to describe which Saga Agent provides which service, or who is responsible for which Saga Agent.

From the discussions above, Saga authorization model is modeled as follows. In the model, we define the *current user* of a Saga Agent as the user who has invoked the Agent. A Saga Agent executes on behalf of its current user. Then, with respect to a Saga Agent o, its current user u, and a service $o.s$, a *service context* σ is defined as the pair:

$$(u, o.s).$$

In Saga authorization model, authorization is given on the basis of service contexts. In other words, service contexts in Saga authorization model correspond to subjects/objects in conventional authorization models. This is a great departure from conventional models. Moreover, in principle, we could contain in a service context arbitrary elements other than the above ones, for example, time variable, predicate on a system state, and we could perform security control according to them. Thus, the concept of a service context is applicable to a wide range of situations.

Now we can see from the above discussions that Saga authorization provides highly flexible and uniform protection in current distributed systems.

4.2 Service Paths in Distributed Environments

In this section, we shall evolve the idea presented in Section 4.1.

In open distributed systems, multiple Saga Agents work in cooperation to achieve some goal. Conventional access control matrix models cannot be fully applicable to such situations.

To discuss this, consider the case shown in Figure 1. Saga Agent o_1 provides a service listTop10TaxPayers, which displays the data about top 10 taxpayers in some expected style. listTop10TaxPayers calls two auxiliary services, namely, getPaidTaxList of Saga Agent o_2 and getNameByTaxPayersNo of Saga Agent o_3. getPaidTaxList of o_2 sorts all the paid taxes in descending order and returns the result together with corresponding taxpayer numbers. getNameByTaxPayersNo of o_3 takes a taxpayer number as an argument and returns the name of the taxpayer. In the following discussion, for the sake of brevity, we mention only services in service contexts.

Now consider the case where a user u_1 and a user u_2 are about to invoke listTop10TaxPayers. Let us assume that u_1 is authorized to know the taxpayer names, the taxpayer numbers, and the paid taxes in the returned list, but assume that u_2 is authorized to know only the taxpayer numbers and the paid taxes. Moreover, suppose that u_1, u_2, and o_1 are not authorized to directly invoke getPaidTaxList and getNameByTaxPayersNo. Consequently, when u_1 invokes listTop10TaxPayers, both of getPaidTaxList and

[2] For a full description of a framework of mobile agents, see [4, 8].

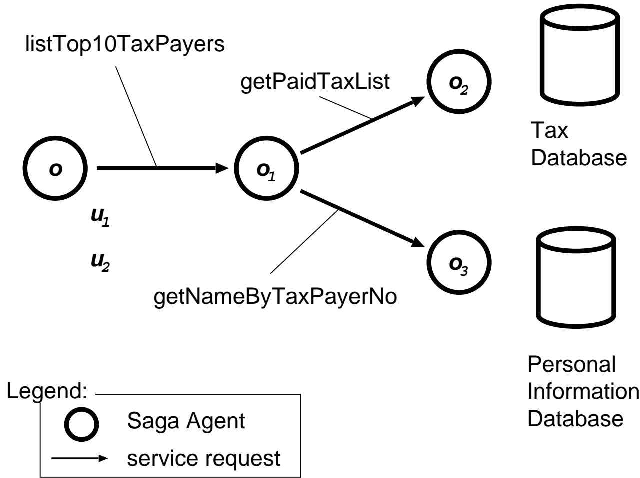

Figure 1: **Service Relationships**

getNameByTaxPayersNo should be allowed to execute, but when u_2 invokes listTop10TaxPayers, nothing but getPaidTaxList should be allowed.

Traditional authorization models can only express simple client/server relationships. Then, the most common solutions to the problem would be as follows:

1. We grant the authorization to invoke both of getPaidTaxList and getNameByTaxPayersNo to the service listTop10TaxPayers. When the Saga Agent o_1 receives a request for listTop10TaxPayers, o_1 invokes either of or both of getPaidTaxList and getNameByTaxPayersNo in compliance with security policy.

2. According to each user's own authorization, we decompose listTop10TaxPayers into services so that the user may be authorized to invoke some of them.

Nevertheless, these *solutions* have a fatal drawback that applications must provide protection by themselves because the security models and mechanisms alone cannot deal with the problem above. To put it another way, applications are imposed restrictions on not only by their own semantics, but also by security policies. Thus, for instance, if u_2 is authorized to call getNameByTaxPayersNo sometime later, o_1 must be rebuilt (possibly from scratch) to incorporate the change in it. To make matters worse, it is nearly impossible that the solutions above apply to more complicated situations, say, getPaidTaxList or getNameByTaxPayersNo further invokes services of other Saga Agents.

Therefore, authorization models in distributed systems must be able to represent, independently of application semantics, situations where agents or objects work in cooperation as the example above shows. For that purpose, we have only to be able to *set authorizations based on sequences of invoked services*. To demonstrate the effectiveness of this approach, consider again the problem depicted in Figure 1. With the approach, we can readily solve the problem as shown in Table 1 ('→' stands for 'invokes' relation). We should note how simply it can be solved.

We are now ready to state this idea more formally. In Saga authorization model, we define a *service path* as a sequence of service context invocations:

$$\sigma_1\sigma_2\ldots\sigma_i$$

where σ_1 invokes σ_2, which further invokes σ_3, ..., and finally σ_{i-1} invokes σ_i. In this paper, a service path is denoted α. Additionally, a *level* is associated with a service path. The level of a service path is defined as i if the service path consists of i service contexts.

Then, in Saga authorization model, authorization to invoke a service context σ is not given to a principal as usual, but *to a service path* $\sigma_1\sigma_2\ldots\sigma_i$. To state the situation simply and clearly, a pair of a service path α and a service context σ:

$$(\alpha, \sigma)$$

is called a *request pair* in Saga authorization model. Then, when a request routed along α to call σ is allowed, we say that (α, σ) is authorized.

The concept of service paths is one of the key features of Saga authorization model and is never found in other traditional authorization models. The introduction of service paths makes it possible to uniformly synthesize the designs discussed in Section 4.1 and 4.2 into Saga authorization model and to realize flexible and powerful protection in distributed systems.

4.3 Delegation in Saga Authorization Model

With respect to a service path, there is another important aspect that we should not miss: namely, it is trust relationships, in particular, *delegation* of privileges expressed in a service path.

To take a closer look at this, let us consider Figure 1 again. Note that o_1 by itself is not authorized to execute getNameByTaxPayersNo, but authorized if and only if u_1 requests o_1 to execute listTop10TaxPayers. That is to say, it is the request of u_1 for listTop10TaxPayers that brings o_1 the authorization to invoke getNameByTaxPayersNo, which o_1 by itself does not have. This implies that the request of u_1 for listTop10TaxPayers is equivalent to the agreement of u_1 on o_1's execution of getNameByTaxPayersNo: In other words, u_1 *delegates* some of its privilege to o_1 through the request. Owing to the expressive power of service paths, Saga authorization model can clearly represent such situations. Furthermore, we can evolve the idea by explicitly dispatching service paths. See [17] for further details.

Needless to say, it is not true that all requests immediately imply delegation in Saga authorization model. However, Saga authorization model can naturally express delegation if necessary, which cannot be easily dealt with by traditional authorization models.

The notable concept of service paths is of great benefit to us, although, it is a little formidable task to implement them securely in open distributed systems. Therefore, next in Section 5, we thoroughly discuss the security mechanism to realize Saga authorization model.

5 Security Mechanism

In this section, we present the security mechanism of Saga Security System, highlighting its outstanding features,

		service	
		getPaidTaxList	getNameByTaxPayersNo
client	$u_1 \rightarrow$ listTop10TaxPayers	allowed	allowed
	$u_2 \rightarrow$ listTop10TaxPayers	allowed	denied

Table 1: **Authorizations given to Sequences of Services**

access tokens, *access control vectors*, and *Security Monitors* (Figure 2).

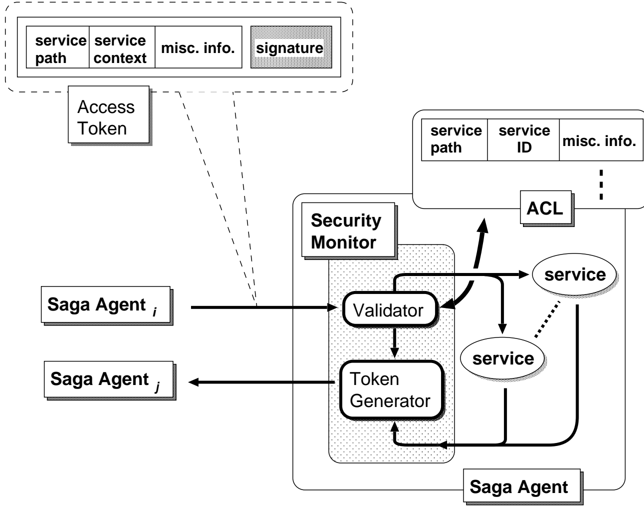

Figure 2: **Security Mechanism of Saga Agent**

5.1 Access Token

To implement Saga authorization model, we cannot overemphasize the importance of ensuring the authenticity and integrity of a request pair (α, σ), in particular, those of a service path α. These are achieved in Saga Security System with *access tokens*, combined with *public key cryptosystems* [5]. An access token of a Saga Agent is constructed by the *token generator* of the Agent.

In public key cryptosystems, a key for encryption and one for decryption are not identical. An encryption key is called a *public key* and can be made public, on the other hand, a decrptyion key is called a *private key* and must be kept secret. One of the important features of public key cryptosystems is that they can generate a *digital signature* on a message, with which we make sure that the identity and the content of the message have not been compromised. In the rest of the paper, K and K^{-1} denote a public key and the corresponding secret key, respectively. In addition, we write $\{M\}_K$ to mean that a message M is encrypted (or decrypted) with a key K.

Using public key technology, Saga Agents submit request messages via *access tokens*. To discuss the structure of an access token, let us consider the case where a Saga Agent o_i sends to a Saga Agent o_{i+1} a request (α_i, σ_{i+1}) where $\alpha_i = \sigma_1\sigma_2 \ldots \sigma_i = (u_1, o_1.s_1)(u_2, o_2.s_2) \ldots (u_i, o_i.s_i)$ and $\sigma_{i+1} = (u_{i+1}, o_{i+1}.s_{i+1})$. τ denotes an access token and let τ_i be the access token corresponding to (α_i, σ_{i+1}). For convenience, τ_0 is supposed to be $(u_1, o_1.s_1)$.

Now τ_i is defined as follows:

$$\tau_i = \{\alpha_i, \sigma_{i+1}, \Pi_i, \Theta_i\} \quad (i = 1, \ldots)$$

where

- $\Pi_i = \{I_1, I_2, \ldots, I_i\}$
  I_i includes the constraints specified by o_i, for example, an expiration time of the access token, with which o_i can counter a replay attack [5] on it.
- $\Theta_i = \{\theta_1, \theta_2, \ldots, \theta_i\}$
  Θ_i is defined as the set of the signatures, $\{\theta_1, \theta_2, \ldots, \theta_i\}$, where $\theta_k = \{\theta_{k-1}, \sigma_{k+1}, I_k\}_{K_k^{-1}}$, $k = 1, \ldots, i$. We assume $\theta_0 = \sigma_1 = (u_1, o_1.s_1)$.

α_i, σ_{i+1}, and Π_i are stored in τ_i in plaintext.

The authenticity and integrity of a request pair (α_i, σ_{i+1}) in τ_i is verified in the following manner. Recall that $\alpha_i = \sigma_1\sigma_2 \ldots \sigma_i$ by definition. Then, receiving τ_i, o_{i+1} can immediately generate i tuples from τ_i, $\{\theta_k, \theta_{k-1}, \sigma_{k+1}, I_k\}$ where $k = 1, \ldots, i$. Next o_{i+1} verifies that $\{\theta_1\}_{K_1} = \{\theta_0, \sigma_2, I_1\} = \{\sigma_1, \sigma_2, I_1\}$ and using I_1, o_{i+1} also verifies that the conditions specified by o_1 are satisfied. If all the verifications succeed, similarly o_{i+1} verifies $\{\theta_2\}_{K_2} = \{\theta_1, \sigma_3, I_2\}$ and the conditions in I_2, ..., and eventually $\{\theta_i\}_{K_i} = \{\theta_{i-1}, \sigma_{i+1}, I_i\}$ and the conditions in I_i. Note that a straightforward implementation of a request pair would be vulnerable to the attack of insertion, deletion, and exchange of service contexts of the service path because a service path is a sequence of service contexts[3]. However, we have a set of signatures $\theta_k = \{\theta_{k-1}, \sigma_{k+1}, I_k\}_{K_k^{-1}}$, $k = 1, \ldots, i$, in τ_i and using them we provide protection against the attack.

Now, from these discussions, we see that the structure of an access token and a digital signature allow o_i to specify σ_{i+1} and I_1, but not to alter α_i freely. Of course, intruders cannot compromise the authenticity and integrity of access tokens. Therefore, with access tokens, we can realize Saga authorization model in open distributed environments.

Furthermore, since a service path can represent delegation of access rights as discussed in Section 4.3, an access token for a request pair inherently has an aspect of a delegation token [15]. Then, Π_i of an access token can be regarded as a condition under which an access privilege is transferred.

[3]This kind of vulnerability is similar to that of block cipher[5].

5.2 Access Control Vector

Every Saga Agent has an access control list, ACL, for services implemented in it, and using the ACL, the Agent determines whether a request issued to it is authorized or not. The ACL in a Saga Agent is a list of *access control vectors*. An access control vector consists of three parts: (1) a service path, (2) a service reference (Saga Agent ID and current user ID are not needed here), and (3) miscellaneous information.

When an access token for a request (α, σ) is sent to a Saga Agent, the Saga Agent searches its ACL for an access control vector corresponding to the request. Direct implementation of this search might incur severe performance degradation since we have to compare service contexts of α and those of access control vectors one by one. To avoid this and perform effective search, a Saga Agent sorts its access control vectors in advance and performs binary search over them.

5.3 Security Control by Security Monitor

A service called a *Security Monitor* is associated with every Saga Agent. It is the Security Monitor of a Saga Agent that performs security control over the Saga Agent.

When a Saga Agent o receives an access token for a request (α, σ), its Security Monitor m verifies the token in the way as discussed in Section 5.1. All accesses to a Saga Agent and accesses from the Agent cannot be bypassed, but are mediated and controlled by the Security Monitor of the Agent. Especially, security of all accesses to a Saga Agent is controlled by the *validator* of the Security Monitor of the Agent. Therefore, we can regard the Security Monitor of a Saga Agent as a *reference monitor* [10] for the Agent. This way, while a Saga Agent travels over an open distributed system, its security can be controlled by the Security Monitor integrated with it.

Furthermore, note that a Saga Agent is highly structured as depicted in Figure 2. Roughly speaking, we can clearly decompose a Saga Agent into two parts: (1) its Security Monitor, and (2) services that implements application semantics. As mentioned in Section 4.2, Saga authorization model is independent of application semantics, and this is true of the structure of a Saga Agent. Therefore, either of the Security Monitor and services of a Saga Agent are little affected by modification of the other, and can be developed without concerns about the other.

6 Related Work

This section examines the previous work relevant to ours.

Java is an object-oriented system developed by Sun Microsystems [7]. Source codes of Java are compiled into *bytecodes*, which are platform-neutral intermediate codes. Executable bytecodes, called *applet*, can be retrieved and executed by WWW clients. Consequently, security is one of the greatest concerns from the beginning of the design of Java. Unfortunately, although various notable security mechanisms are incorporated into Java, the approach is a little ad hoc because of the lack of formal security models [11].

In order to take advantage of encapsulation of object-oriented systems, the authorization models where users are given authorization on methods, not on primitive operations, have been an active area of research in recent years [1, 2]. Nevertheless they are not so uniformly modeled as Saga authorization model is. Moreover, surprisingly few models addressed current open distributed environments.

Rabitti et al. [14] developed a sophisticated authorization model on ORION object-oriented database. In the model, exploiting relationships among subjects, objects, and access modes, we can derive implicit authorizations from explicitly specified authorizations. However, the model could not appropriately deal with the situations where agents or objects work in cooperation in distributed systems. In other words, the model is orthogonal to Saga authorization model and it is possible to integrate both of them.

PCM (Path Context Model) [3, 12] takes into account potentially insecure distributed environments and can perform access control in terms of access path. Although PCM does not model semantics of authorization, it can express a wide variety of conditions of network environments, for instance, network domains and encrypted channels. It is desirable to accommodate such expressive power into our model in the future.

7 Conclusion and Future Work

Since current open distributed environments are expected to prevail, it is of critical importance to establish security of a proper level in them. Nevertheless, in conventional "secure" systems, it is often the case that these environments are not taken into account. Additionally, traditional access control matrix models cannot be fully applicable to such environments.

In the paper we have discussed so far an overview of Saga Security System, a security architecture in such environments, highlighting Saga authorization model and its security mechanism. The model is uniform and flexible, and is appropriate for advanced computing models such as object-oriented systems and agent systems, in open distributed systems. Additionally, we have presented the security mechanism of Saga Security System, which provides protection of a Saga Agent while it travels over an open distributed system. Hence, the security model and mechanism of Saga Security System realize in current open distributed systems a promising security architecture, which has been strongly required but cannot be provided by traditional systems.

Before concluding the paper, we should make a remark about our current research status. We are now developing a prototype version of Saga Security System, and at the same time, doing research on how Saga authorization model can be well adapted to a specific object-oriented system, and how the model can enforce a security policy of an enterprise organization, especially, how it can enforce multiple security policies of federated systems.

Acknowledgments

We would like to thank other members of Saga Security System Project, Norihiko Kameda, Kiyoshi Une, Satoshi Yoshida, Atsuki Tomioka, Keisuke Yamaguchi, and Takeharu Kato for useful discussions and comments.

References

[1] R. Ahad, J. Davis, S. Gower, P. Lyngback, A. Marynowski, and E. Onuegbe. Supporting access control in an object-oriented database language.

In *Proc. 3rd International Conference on Extending Database Technology (EDBT)*, pp. 184–200, 1992.

[2] E. Bertino and P. Samatari. Research issues in discretionary authorizations for object bases. In *Proc. OOPSLA'93 Workshop on Security for Object-Oriented Systems*, pp. 183–199, 1993.

[3] W. H. Boshoff and S. H. Solms. A path context model for addressing security in potentially non-secure environments. *Computers & Security*, 8(5):417–425, 1989.

[4] D. Chess, B. Grosof, C. Harrison, D. Levine, and C. Parris. Itinerant agents for mobile computing. Technical Report RC 20010, IBM Research Division, T.J. Watson Research Center, Mar. 1995.

[5] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Co., Reading, MA, 1982.

[6] Department of Defense. DoD trusted computer system evaluation criteria. Technical Report DoD 5200.28-STD, DoD Computer Security Center, Fort Meade, MD, Dec. 1985.

[7] J. Gosling and H. McGilton. The Java language environment: A white paper. Technical report, Sun Microsystems, 1995.

[8] C. G. Harrison, D. M. Chess, and A. Kershenbaumn. Mobile agents: Are they a good idea? Technical Report RC 19887, IBM Research Division, T.J. Watson Research Center, Oct. 1994.

[9] Information-technology Promotion Agency (IPA), Japan. Research report on security architecture and automatic user authentication in downsizing environments. Technical Report 159, Information-technology Promotion Agency (IPA), Japan, Mar. 1996. In Japanese.

[10] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium of Information Sciences and Systems*, pp. 437–443, Mar. 1971.

[11] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes and Antidotes*. John Wiley & Sons, Inc., 1997.

[12] M. S. Olivier and S. H. Solms. Building a secure database using self-protecting objects. *Computers & Security*, 11(3):259–271, 1992.

[13] G. J. Popek and C. S. Kline. Encryption and secure computer networks. *ACM Comput. Surv.*, 11(4):331–356, Dec. 1979.

[14] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Trans. Database Syst.*, 16(1):88–131, Mar. 1991.

[15] K. R. Sollins. Cascaded authentication. In *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 156–163, Apr. 1988.

[16] M. Soshi. The design and implementation of the Security Agent mechanism. In *The Collection of Papers in the 15th Technical Presentations*, volume 15, pp. 225–234. Information-technology Promotion Agency (IPA), Japan, Oct. 1996.

[17] M. Soshi, T. Kato, and M. Maekawa. Proxies in distributed systems: Flexible protection realized by simple and dedicated agents. Submitted for publication, 1997.

[18] M. Soshi and M. Maekawa. A new authorization model and its mechanism using service paths in open distributed environments. To appear in the Proceedings of IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, Sept. 1997.