DESIGN OF THE SAGENT SECURITY FRAMEWORK FOR JADE

Vandana Gunupudi
Department Computer Science and Engineering
University of North Texas
Denton, TX 76203
email: gunupudi@cs.unt.edu

Stephen R. Tate
Department Computer Science and Engineering
University of North Texas
Denton, TX 76203
email: srt@cs.unt.edu

ABSTRACT
In this paper, we present the design of SAgent, a general-purpose mobile agent security framework. SAgent is designed for comprehensive protection of mobile agent computations and data in potentially hostile environments and works with the JADE (Java Agent DEvelopment) platform [1], a FIPA-compliant multi-agent environment. Using good software engineering design techniques of software reusability and abstraction, SAgent allows agent protection protocols and applications to be developed independently of each other. To accomplish this, a clean conceptual framework is presented which encapsulates in several general class interfaces the common security functionality required by secure agent applications. Since SAgent is designed to generically protect the computations of mobile agent applications, we provide implementations of two secure multiagent protocols that protect the *confidentiality* of agent data as well as implementations of four methods that protect the *integrity* of mobile agent data. Experimental results showing the feasibility of these methods are available in separate publications [2] [3]. The goal of SAgent is to provide a framework where proposed theoretical techniques can be used and experimentally evaluated. SAgent allows a new security provider to implement and experiment with new techniques for protecting mobile agents in a well-defined manner and is generic enough to support both software-based and hardware-based protections.

KEY WORDS
Software Development, Software Reuse, Distributed Agents, Mobile Agent Security, Frameworks for Mobile Agents

1 Introduction

SAgent is a general-purpose agent security framework for the JADE platform that is designed to protect the computations of mobile agent applications in potentially hostile environments. The JADE (Java Agent DEvelopment) platform [1] is a popular mobile agent platform that enables the development and deployment of multi-agent applications using Java, and conforms to FIPA standards for software agents. SAgent supports the development of secure mobile agent applications, where data is protected from compromised or malicious hosts. Various theoretical methods [4] [5] have been proposed to protect agent data and computations from malicious hosts and the aim of SAgent is to bring these theoretical explorations into practice so that various methods can be evaluated for their practicability.

The SAgent framework is based on a general and widely applicable model of agent computation. Specifically, an agent owner, also called the *originator*, creates mobile agents to perform some task on her behalf. After creating the agents for some specific purpose, the originator sends them out to visit various remote hosts, where the agents perform computations and actions on behalf of the originator. The agents carry data (state) with them, can accept inputs from the hosts they are visiting and from other agents, and agent computations can provide results to hosts and to other agents. The agent computations at each host can be protected in SAgent and this functionality is modeled as a 3-input, 2-output function, with the three inputs being the input from the host, the input from the agent and the agent state. The outputs are the updated agent state and output for the host. When the agents return home, the originator accesses the agent state to determine the results of the remote computations and interactions.

The functionality required by the agent can be viewed as two distinct pieces: one part required for computations at visited hosts and other piece of the functionality required only by the originator. This concept mirrors that of public key cryptography where a public key is known to everyone and the corresponding private key is kept secret by the owner of the key. In the mobile agent paradigm, the originator keeps some functionality private, while incorporating within the agent public functionality that it requires to perform computations on visited hosts. The private information is necessary only for the final interpretation of the results and does not need to be available in any way during the agent's travels. This separation of public and private functionality is the key abstraction in SAgent and our design separates the public and private functionality of the agent into distinct pieces.

SAgent also distinguishes between a mobile agent *application* and a mobile agent protection technique or *security protocol*. Different applications have different security requirements and some applications may not even require any protection. With this in mind, SAgent allows developers to create agent applications and security protections independently of each other. This creates two different perspectives of SAgent – one of a programmer that creates protection techniques in SAgent and the other of an application developer. In order to allow for data in various application-dependent formats, SAgent defines a generic data format usable within SAgent. This generic format, while not secure, is also not specific to any application. This allows application and protocol developers to develop their protection methods and applications using data formats of their own choosing. The application developer supplies routines to convert application specific data into a generic, intermediate format and the protocol developer supplies routines to translate this generic format into a secure format usable within SAgent. Data can then be operated on in this secure format by going through methods in the generic public interface, which resolve to the appropriate protocol-specific methods.

This paper focuses primarily on the design of the SAgent framework, as well as some implementation details. Results from an extensive set of experimental tests on these protocols are available in separate publications [2] [3]. All the software described in this paper, including the SAgent framework, protocol implementations, and sample applications, is freely available for download [6] and redistribution under the LGPL license.

2 Relation to Prior Work

We believe that SAgent is unique in bringing into practice a software framework for protecting agents from malicious hosts with rigorous security requirements. Much of the previous work deals with securing communication or infrastructure, or providing agent authentication. For example, Poslad and Calisti address the problem of securing the basic FIPA agent platform infrastructure, including integrity guarantees for communication with the agent management system and directory facilitator [7]. Zhang *et al.* also consider security issues in FIPA, providing mechanisms for authentication, secure communication, and resource monitoring [8]. In their paper, Zhang *et al.* do refer to the problem of securing agents from malicious hosts, but techniques for such protection were not widely known in 2001, resulting in the dismissal of this problem saying “a lack of effective methods prevented our architecture from dealing with the risk from a malicious agent platform” [8]. Finding theoretical solutions to these problems has allowed for the possibility of our SAgent work, and so we summarize these results next.

In 1999, Bennet Yee [9] proposed a comprehensive hardware-based solution to protect mobile agents that provides a trusted execution environment that runs within a secure coprocessor. This secure environment allows Java agents to run securely and migrate among unmodified Java interpreters that run within these secure coprocessors [9]. For a while it was thought that it was not possible to protect the agent code from malicious hosts without the use of trusted hardware. The work of Sander and Tschudin [10] first showed that software-only techniques were possible, at least for a class of problems that supported a homomorphic encryption scheme. They provided a non-interactive protocol for solving the basic problem of Computing with Encrypted Functions (CEF) for rational functions. Subsequent to Sander and Tschudin’s ground-breaking paper, several results extended the class of functions that could be protected, culminating in a result of Algesheimer *et al.* showing that any polynomial-time computation could be protected without any trusted hardware being required [4]. In SAgent and in this paper we refer to this technique as the “ACCK protocol,” after the initials of the authors of the paper. The ACCK protocol is based on Yao’s encrypted circuit and two-party Secure Function Evaluation (SFE) protocol [11], and it requires the participation of a trusted third party (TTP) which performs certain decryptions for the remote hosts. The model was made less restrictive by Tate and Xu, who demonstrated how a modified protocol could be performed using threshold cryptography and secret sharing instead of a trusted third party [5, 12]. We refer to this technique as the “TX protocol,” and note that no trust assumptions are required other than assuming that the number of dishonest parties is limited by a threshold value which is a parameter to the protocol. In SAgent, we do not invent new protection techniques, but rather unify these previously-presented techniques under a common framework and provide implementations of the ACCK and TX protocols.

We also make the important point that the protocols supplied with SAgent have rigorous security guarantees, in contrast to some previous work that uses heuristic techniques of code obfuscation or simple integrity tests. For example, Page *et al.* provide an interesting technique for detecting modifications to agent code, but deeper platform tampering (for example, with the hosting software or the Java Virtual Machine) cannot be either detected or prevented [13]. By contract, both the ACCK and TX protocols make no assumptions about a properly functioning platform or JVM.

3 Sample Applications

SAgent is distributed with two sample applications, which we describe in this section. These applications exercise various parts of the SAgent framework, so understanding these applications can make concrete some of the abstract notions that we describe in the following sections. Both applications are in the general area of e-commerce agent applications.

Maxbid application. In this application, the originator wants to sell an item and sends one or more agents out to visit remote hosts to gather bids for an item the originator wishes to sell. Each host gives its bid and the agent keeps the *maximum* bid it has seen so far in its state. We would like to keep the current maximum bid secret so that hosts cannot cheat by bidding slightly over the current maximum. When the agents return to the originator, the originator decodes each agent’s maximum bid and computes the overall maximum bid. The protected function in this application is a comparison of two integer values (current maximum bid and bid offered by current host). In this application there is no unprotected agent input and there is no output to the host.

BuyAgent application. This application is slightly more complex than the Maxbid application, using all three of the protected functionality inputs and providing the host with one of the outputs. In this application, the originator wishes to buy an item, so the agent maintains the *minimum* bid it has seen so far. Furthermore, the minimum bid is time-stamped, with the time being provided as the unprotected agent input (unprotected since the current time is not sensitive information). In addition, the originator programs a threshold price called the "buy now" value into the agent, whereby, if any host offers a bid lower than the threshold, the agent provides the host the originator's credit card number to immediately buy the item. Clearly, both the "buy now" value and the credit card number are sensitive pieces of data, and should be protected. The function in this application is a comparison of two integer values, with the evaluation resulting in an update of the minimum value and the corresponding timestamp. The protected function also provides a comparison with the "buy now" threshold, revealing the credit card number only if the threshold value is met.

4 Core Functionality of SAgent

Using sound principles of object-oriented design, SAgent specifies various general interfaces which are extended by security providers to design protocol-specific implementations. As described earlier, SAgent separates the private and public functionality of the agent into distinct pieces, which correspond roughly to the agent originator's secret information and the information required by the agent for performing computations at remote hosts. The core of our design, therefore, includes secure private and public interfaces, along with a secure data interface. In the rest of this section we describe these core interfaces in brief (refer to our previous work [14] for details), and describe the functionality they specify.

An application designer calls a provider-supplied constructor with appropriate parameters in order to set up the agent application, and from that point on all operations are performed through the generic interfaces. To illustrate this, we point out that SAgent supports both configuration files and command-line parameters that allow protocols and protocol parameters to be changed on the fly. For example, to change from the TX protocol to the ACCK protocol in the Maxbid application, starting the originator can be done with the following command:

```
java jade.Boot -container \
 Orig:MaxbidOriginator(SAgent.protocol:ACCK)
```

While there is a conditional test (an `if` statement) in the MaxbidOriginator code to call the appropriate constructor, not a single line of code in the MaxbidMobileAgent class depends on the protocol being used, and this on-the-fly change of security provider is completely transparent to the mobile agents and remote hosts.

The `SecureFnPrivate` interface encapsulates the private functionality in our mobile agent model, defining the methods for interacting with the agent that are restricted to the agent originator. This functionality includes methods for setting the initial state of the agent and for decoding the final state of the agent when the agent returns home. The agent state can be updated at visited hosts but the state need not be decoded by any entity other than the originator. Methods required by the agent for performing computations at remote hosts correspond to the public functionality and are defined in the `SecureFnPublic` interface which encapsulates the functionality required by the agent to perform protected computations at remote hosts. The public functionality provides means for encoding the agent's input (obtained from unprotected agent computations) as well as the ability to interact with a visited host and encode the host's input in an appropriate manner.

4.1 Agent-Host interactions

Below, we describe how the public functionality of an agent, via the `SecureFnPublic` interface, is used to perform computations at a visited host, while preserving the security of the agent data. The details of the interactions are illustrated in Figure 1 as well.

Initialization. Once the originator creates a mobile agent and initializes it with the public part of the protected functionality, the mobile agent should call the *initializeSecurity* method to initialize any structures that are tied to the mobile agent. In some protection techniques, this might not do anything, but some techniques (such as the TX protocol) require some specific per-agent initialization. However, whether or not a protocol requires this step is not something the application developer should consider, and should simply call this method as the first step of the mobile agent's actions. Following this initialization, the mobile agent can use the *secureMove* method to transfer itself to the first remote host.

Encoding of agent and host inputs. Upon successful contact with the host (through the host agent), the agent supplies its public part to the host and asks the host for its input. The public part of the agent includes the *encodeHostInput* method, which is then used by the host to encode its input in an appropriate format. Specifically, this involves reading in the host's input in a generic form and converting it into an object of type `SecureFnData`, where the specifics of this transformation depend on the particular protocol which implements this interface. Only the encoded input is returned from the host to the agent, which is

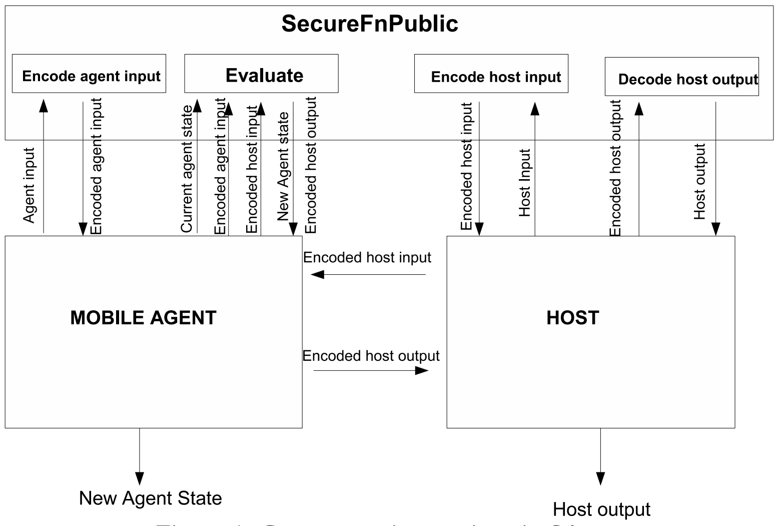

Figure 1. Component interactions in SAgent

what enforces that the host data is used only as specified by the particular agent functionality. Similarly, if the agent has unprotected computation results that need to be included in the protected computation, then it encodes its input using the *encodeAgentInput* method.

Evaluation of the protected function. Once the agent obtains the host and agent inputs in a secure format, it calls the *evaluate* method to perform the protected computation. This method updates the agent state and returns a `SecureFnData` object representing the output to be provided to this host, which can be converted to plaintext using the *decodeHostOutput* method. This evaluation obviously depends on the specifics of the agent application, but the evaluation is protected in a uniform way by the particular security provider. This is one of the key features of SAgent, whereby the evaluation function operates only on *secure* objects.

Agent movement. Once a mobile agent has completed its computation at a particular host, it decides to either migrate to another host or to return home. Movement is requested through the *secureMove* method in the `SecureFnPublic` interface. When the agent returns home, the final state is communicated to the originator as a `SecureFnData` object, and the originator can decode this using the *decodeAgentState* method in the `SecureFnPrivate` object that it has retained from when it created the agents.

The key to all of the mobile agent computations is that computations are only performed on `SecureFnData` objects, and the privacy and/or integrity of these operations is guaranteed by the security provider and the security protocol. Any protected information stored in the agent state can be operated on through the *evaluate* method, but can not be decoded or decrypted except through the `SecureFnPrivate` object, which only the originator possesses. Sample code showing typical calling sequences is given in Section 5.2.

4.2 Data in SAgent

SAgent is designed to protect the security of the mobile agent data. When data are operated upon within SAgent, the data are in a secure format, i.e., only *protected* data objects are allowed within the protected execution. To accommodate diverse application-specific data formats, SAgent defines an application-independent intermediate format that can be used within SAgent. This is just a binary representation of the data, and makes it simple for application-developers and security providers to develop their products independently of one another.

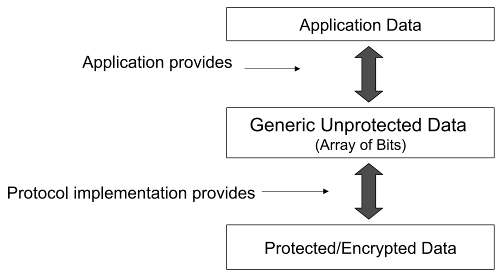

Figure 2. Representation of data in SAgent

Figure 2 shows the different formats that data can be represented in in the SAgent framework. Data comes into the SAgent framework in an application-dependent format, is converted by an application developer-supplied method to a generic unprotected format, and is then converted into the secure format by routines given by the security provider. All computations with data in SAgent are performed in this secure format. When data is to be returned to the application, it is converted to a generic format (by the protocol) and then passed to the application. The application then converts this generic format to a form specific to the application. This abstraction and automatic conversion between unprotected and protected forms is one of the key ideas of SAgent, which gives the framework an elegant way of separating application developer and security provider interests.

5 Programming with SAgent

The previous sections provided a high-level overview of SAgent, showing how protocol-issues and application-issues are kept separate to allow for independent development. In this section we give more details on the tasks that must be performed in order to provide a new protection technique within the SAgent framework, or to write an application in the framework. We will use the protocols and applications provided with SAgent to highlight various steps in the development process.

5.1 Making secure protocols with SAgent

SAgent allows programmers to develop agent protection techniques which may be used by a wide variety of applications. Implementations of protocols for SAgent may have different characteristics, and a user can select a protocol which satisfies her needs. In this section, we describe the steps a security provider must take when programming a new protocol.

Step 1: *Decide how protected functions and data are represented.* Typically data will be encrypted, and an application needs to be able to describe the protected functionality to the protocol. In software-based protections for arbitrary polynomial-time functions, such as ACCK and TX, the function is typically represented by a boolean circuit and the data is represented with random multi-bit "signals" for the input wires. For software-based techniques using homomorphic encryption for smaller classes of functions, such as the technique of Sander and Tschudin [10], the data could simply be the ciphertext after applying the homomorphic encryption scheme, and the function representation would depend on the technique being applied. For hardware-based techniques, the data representation could similarly be a simple encryption, and the function description could be as straight-forward as a Java class.

Step 2: *Determine what needs to be provided to hosts in order for them to encode their inputs, and create the encodeHostInput and encodeAgentInput methods.* Inputs will need to be encoded when the agent is traveling, so any data necessary for this operation must be identified and placed in the public part of the agent. For example, the ACCK protocol uses a trusted third-party (TTP) to decrypt appropriate input signals for the host, and so included with the public part of the agent is the set of all possible signals, encrypted with the TTP's public key. With a homomorphic encryption scheme, this could be the public key for the encryption scheme. With hardware-based protections, this might not require any information, as the host can pass unencrypted data directly into the protected execution environment.

Step 3: *Make an evaluate method that uses the data and function representations from Step 1.* Once the decisions about representation are made in Step 1, this step is typically fairly simple. The gate-by-gate evaluation required by circuit techniques is supplied in the same package as the circuit definitions.

Step 4: *Determine what needs to be provided to hosts in order for them to decode their outputs.* The evaluation phase produces `SecureFnData` protected data, so information must be included in the public part of the agent in order for these results to be rendered intelligible to the host. For circuit-based techniques, this is the signal-decoding information for the output wires. For homomorphic encryption schemes, this could be the agent identifier of a trusted party that can decrypt the output for the host. Similar to the inputs, hardware-based protections will most likely not need any information, since plaintext data can be passed out of the protected execution environment.

5.2 Making secure applications with SAgent

An application developer wants to concentrate on the application they are designing, and not worry about the particulars of any specific agent protection method. The application developer is responsible for creating an originator agent and the mobile agent(s). The application developer could also create a host agent for use and/or adaptation by service providers, or could simply specify the data required and let the service providers create their own `HostAgent` implementations. Below, we describe the basic steps required of an application developer.

Step 1: *Decompose the application into protected and unprotected parts.* In order to remove undue complexity which could lead to security problems, it is a good idea to protect only the computations and data that truly require protection, such as the credit card numbers and bid values in our sample applications. This is also wise from an efficiency standpoint, since protected computations can be noticeably less efficient than unprotected computations.

Step 2: *Describe the protected computation in an appropriate representation.* This is the one step that can potentially depend on the protocol, as the representation may be different with different protection schemes. Our supplied protocols use boolean circuits, as do other general techniques, so this representation can be used with a wide range of protection techniques without regard for the details of the protection.

Step 3: *Create an originator agent/mobile agents.* The originator must call the constructor for the `SecureFnPrivate` implementation of the chosen protection scheme, passing the protected function representation created in Step 2. The originator can then retrieve the public part of the protected function, create mobile agents that carry this public part, dispatch the agents and wait for them to return. Sample code from our applications showing these steps is shown below:

```
// Encode the host input
SecureFnData hostInput =
                (SecureFnData)hostAgent.askHost(publicPart, myID);

// Evaluate the inputs
SecureFnData hostOutput =  publicPart.evaluate(null, hostInput);
```

Maxbid application

```
// Encode the host input
SecureFnData hostInput =
                (SecureFnData)hostAgent.askHost(publicPart, myID);

// Encode the agent input
SecureFnData agentInput = publicPart.encodeAgentInput(
                new Long(System.currentTimeMillis()), thisAgent);

// Evaluate the inputs
SecureFnData hostOutput = publicPart.evaluate(agentInput, hostInput);

// Return output to host
hostAgent.resultToHost(hostOutput, myID);
```

Buyagent Application

The application developer must of course also program the unprotected functionality in the mobile agent.

As can be seen in this section, the only substantial work required for an application developer to add security protections to her application is to provide a representation of the protected functionality in a general form understood by the chosen protection method.

6 Conclusions

This paper presents the design of the SAgent security framework that we developed for the JADE multiagent platform. We described the core interfaces and showed how each of these interfaces can be used to create protocol-specific instantiations. SAgent allows independent development of agent applications and security protocols, and we discussed in detail how to create new agent applications and security protections in SAgent. We also provided the first software implementations of the ACCK [4] and the TX [5] mobile agent privacy protocols showing the feasibility of these software-only protections within our framework. The generic applicability of SAgent can also be seen in our other publications [2] [3] where we provide experimental results and show how various agent security protocols can be seamlessly integrated into SAgent. Our future work involves exploring the inclusion of support for hardware-based protections in SAgent, and we believe that such an implementation would be straight-forward in a system that provides verifiable Java environments through hardware-based attestation.

References

[1] JADE, Documentation and Resources. Available at: http://jade.tilab.com/.

[2] V. Gunupudi and S.R. Tate. Exploration of Data Integrity Protection in SAgent. In *Proc. of First International Workshop on Privacy and Securi ty in Agent-based Collaborative Environments (PSACE)*, 2006.

[3] V. Gunupudi, S.R. Tate, and K. Xu. Experimental Evaluation of Security Protocols in SAgent. In *Proc.of First International Workshop on Privacy and Securi ty in Agent-based Collaborative Environments (PSACE)*, 2006.

[4] Joy Algesheimer, Christian Cachin, Jan Camenisch, and Gunter Karjoth. Cryptographic security for mobile code. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 2–11, 2001.

[5] Stephen R. Tate and Ke Xu. Mobile agent security through multi-agent cryptographic protocols. In *Proc. of the 4th International Conference on Internet Computing (IC 2003)*, pages 462–468, 2003.

[6] SAgent software available at:. http://cops.csci.unt.edu/sagent/, 2006.

[7] Stefan Poslad and Monique Calisti. Towards improved trust and security in FIPA agent platforms. In *Autonomous Agents Workshop on Deception, Fraud, and Trust in Agent Societies*, 2000.

[8] Min Zhang, Ahmed Karmouch, and Roger Impey. Adding security features to FIPA agent platforms. In *Mobile Agents for Telecommunicatoins Applications (MATA)*, 2001.

[9] B. Yee. A sanctuary for mobile agents. *Secure Internet Programming, LNCS*, 1603:261–273, 1999.

[10] T. Sander and C.F. Tschudin. Protecting mobile agents against malicious hosts. *Mobile Agents and Security, Lecture Notes in Computer Science*, 1419:379–386, 1998. Ed. G. Vigna.

[11] A.C. Yao. How to generate and exchange secrets. *Proc. of the 27th IEEE Symposium on Foundations of Computer Science(FOCS)*, pages 162–167, 1986.

[12] Ke Xu and Stephen R. Tate. Universally composable secure mobile agent computation. In *Proceedings of the 7th International Conference on Information Security (ISC)*, pages 304–317, 2004.

[13] John Page, Arkady Zaslavsky, and Maria Indrawan. Countering security vulnerabilities in agent execution using a self executing security examination. In *Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1486–1487, 2004.

[14] V. Gunupudi and S.R. Tate. SAgent: A Security Framework for JADE. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*, 2006.