SAgent: A Security Framework for JADE*

Vandana Gunupudi Stephen R. Tate
Dept of Computer Science and Engineering
University of North Texas
Denton, TX 76203

Abstract

This paper presents SAgent, a general-purpose mobile agent security framework that is designed to protect the computations of mobile agent applications in potentially hostile environments. SAgent works with the JADE (Java Agent DEvelopment) platform [4], a FIPA-compliant multi-agent environment. SAgent supports modular and mostly orthogonal development of agent protection techniques and secure agent applications, so protocols and applications can 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. Furthermore, implementations are provided for two secure multi-agent protocols, and we give experimental results showing the feasibility of these protections. While a few other research projects have examined protocols and techniques for protecting agents, these have been theoretical explorations. SAgent's goal is to bring these theoretical techniques into practice so that they can be experimented with and used, in the framework of a design generic enough to support both software-based and hardware-based protections. The abstractions are clean, giving a well-defined way for a new security provider to implement and experiment with new techniques for protecting mobile agents.

*This research is supported in part by NSF award 0208640.

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 [4] 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 rogue or compromised hosts. For example, a shopping agent could carry a credit card number or electronic cash token, while being assured that this protected data would only be revealed if certain conditions are met in the agent computation. While other researchers have developed frameworks that provide solutions for various agent security problems, such as authentication of agents, agent replication, and voting [2, 15], SAgent is concerned specifically with protecting the data that travels with a mobile agent. Strong methods for protecting agent data have been proposed by researchers, including hardware-based techniques [14] and techniques based on clever use of cryptography and distributed protocols [3, 1, 9, 10], but SAgent is the first attempt at unifying these techniques under a common framework, and as far as we're aware provides the first experimental explorations of the proposed agent protections.

In SAgent, methods of protecting agent data are cleanly separated from applications using those protections, so both application and protection technique can be independently considered. A number of general security interfaces are defined, which encapsulate the common security functionality required by secure agent applications. "Security providers" can provide implementations using these interfaces, and SAgent is distributed with two different implementations of software-based protection techniques, one by Algesheimer *et al.* [1] and one by Tate and Xu [10]. While the support for these software-based techniques is clear, the SAgent framework is general enough to support hardware-based protections just as cleanly. While the current lack of hardware to provide such protection makes this impossible to experiment with now, SAgent was designed so that if the current proposals for "trusted platforms" [6, 11] become widespread and useful, then simple support in the SAgent framework would be possible. For convenience, we refer to protection methods as "protocols" in this paper, but this should not be taken to imply that only protections based on cryptographic protocols are supported by SAgent.

The contributions of this work include

- A clean and useful abstraction of data protection in mobile agents;
- A working platform that realizes these abstractions and provides an effective framework for both security providers and agent application developers; and
- Implementation of two software-only security solutions that are usable and practical for protecting small amounts of agent data.

All the software described in this paper, including the SAgent framework, protocol implementations, and sample applications, is freely available for download and redistribution under the LGPL license.

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. When the agents return home, the originator accesses the agent state to determine the results of the remote computations and interactions. In general, agent movements are determined by the agent logic and can be dynamic, and the SAgent framework supports this; however, the two agent-privacy methods distributed with SAgent impose a restriction that the originator creates the itineraries for the agents in advance. To summarize the model, a mobile agent application consists of an originator, one or more mobile agents with specific functionality, and host agents (which provide interactions with a host).

This paper focuses primarily on the design of the SAgent framework, as well as some implementation details, and gives only basic data on the efficiency of the specific security protocols. Results from an extensive set of experimental tests on these protocols are being prepared for separate publication.

The paper is organized as follows: Section 2 presents an overview of SAgent, explains the security model, and discusses previous work. Section 3 presents an overview of the Java Agent Development (JADE) platform. In Section 4, we describe the core design features of SAgent, and discuss how the various components interact. Section 5 discusses the steps required for a security provider or application developer to implement new protocols or applications under SAgent. Finally, in Section 6 we give a few experimental results that demonstrate the efficiency of our implementation.

2 Overview of SAgent

SAgent was designed keeping in mind a basic tenet of software engineering, that definition of proper abstractions is essential for software reusability. The key abstraction behind SAgent mirrors that of public/private key pairs in public key cryptography in which a private key is known only to the owner of the key, while the corresponding public key is available to everyone. Similarly, a mobile agent application has *public* functionality and information that it needs to perform computations at remote hosts and *private* information and functionality that is kept by the originator of the agent and not exposed to other entities. Note that while we refer to the information that travels as "public," this does not mean it is understandable to an outside viewer — encrypted data (ciphertext) can be public even if the corresponding plaintext remains unintelligible. 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. To that end, our design separates the public and private functionality of the agent into distinct pieces.

There are two different views of these pieces in SAgent — one from the point of view of a programmer that develops protection techniques for SAgent and another from the point of view of a developer who writes applications for SAgent. The architecture of SAgent is designed so that the security provider and the application-developer can

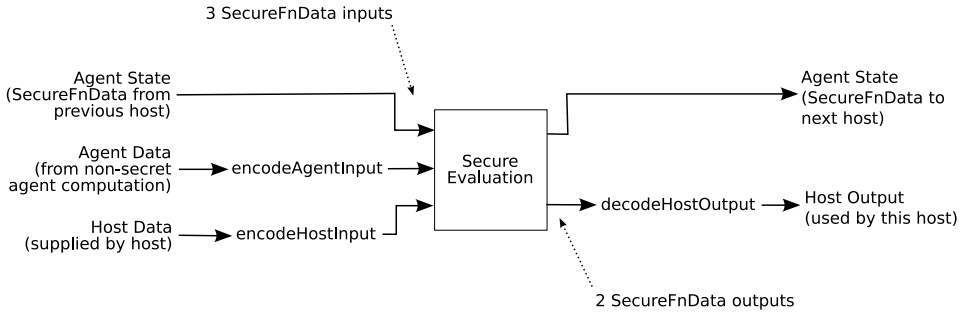

Figure 1: Agent State Update in SAgent

remain unaware of each other and develop protocols and applications independently of one another. Data transfer and usage in SAgent revolves around a generic, intermediate data format that is not secure but also not application-dependent. The application developer is responsible for providing translation routines that convert application-specific data into the intermediate format, and the security provider provides routines which convert this intermediate format into a secure but protocol-specific representation. 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.

2.1 Security Model

Keeping in mind the fundamental secure design principle of *economy of mechanism*, agent applications are decomposed into protected and unprotected computations, with only the sensitive data and computation being run in the protected environment. For instance, a shopping agent may browse through a store using unprotected computations, but could use SAgent to protect the portions of the agent which make purchasing decisions. The protected portion could include very sensitive information, such as payment authorization values, credit card numbers, or electronic cash tokens. The protected agent computation is modeled as a 3-input, 2-output function, as shown in Figure 1. The three inputs are the current agent state, the input from the host, and an input provided by any unprotected computations performed by the agent on that host. The two outputs are an updated agent state and an output that is provided to the current host. Specific security providers guarantee slightly different security properties, but in general the agent state is protected so that it is unintelligible to an outside observer (either an attacker or a malicious host), and the host's input is protected so that it can only be used in a manner consistent with the agent functionality, which can be determined or negotiated as a contract between the originator and the host. In our sample applications we have experimented with encoding credit card numbers both as constants within the protected computation and as part of the agent state. From an efficiency standpoint there is very little difference, so the application designer is free to choose how this private information is included. As an example of an unprotected agent input, one of our applications provides the current time as an input to be used within the protected computation as a timestamp.

2.2 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.

2.3 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 [8]. Zhang *et al.* also consider security issues in FIPA, providing mechanisms for authentication, secure communication, and resource monitoring [16]. 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" [16]. 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 [14] 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 the secure coprocessor [14]. 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 [9] 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 [1]. 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 [13], 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 [10, 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 [7]. By contract, both the ACCK and TX protocols make no assumptions about a properly functioning platform or JVM.

3 JADE

JADE is a multiagent platform, fully implemented in Java, and conforming to FIPA standards for intelligent software agents. JADE exports a group of Java packages for developing agent applications on the platform. For background information, we describes the features of JADE below.

Agent Containers: An Agent Container provides an execution environment, and multiple agents can be active in the same container. The agent platform can be distributed over several machines, and a designated main container serves as the controller of the platform. JADE provides a GUI interface for the main container from which a platform manager can manage all the local and remote containers and agents in the platform, doing things

such as creating and deleting these entities. JADE provides FIPA-compliant system services such as naming, yellow-page service, message transport, agent communication, etc. All entities in the JADE system are agents, so a remote host in the SAgent model is implemented by an agent we call `HostAgent` (or, more precisely, from extensions of the core `HostAgent` class). From the programmer's perspective, a JADE agent is simply an instance of a user-defined agent class that extends the basic `Agent` class.

Agent Behaviours: Agent actions are implemented through extensions of the class `Behaviour`. An agent can have multiple behaviours, each behaviour corresponding to a specific functionality of the agent. The main action function of a behaviour can be executed multiple times. JADE includes a scheduler that schedules the execution of the behaviours in a round-robin fashion. There is no preemption, so every behaviour has to explicitly yield control, which it typically does by reaching the end of its action function. Once the action function returns, the scheduler queries the *done* function of the behaviour to see if it needs to be re-scheduled, and then another behaviour is activated.

Agent Messages: JADE provides an implementation of the full FIPA model. Communication between agents is largely transparent to the developer. Agent messages are defined by the `ACLMessage` class, which contains a set of attributes defined by FIPA. All the messages for a particular agent are put into the agent's private message queue for delivery.

To send a message, a sender specifies the message type using one of the constants defined by FIPA, the receiver's agent ID, and the content of the message. JADE finds the receiver agent and puts the message in the receiver's private queue. The content of an `ACLMessage` can be either an array of raw bytes or a Java object (although the latter is non-standard).

Agent Migration: Agent migration in a behaviour is through the *doMove* method which wraps the Java object and moves it to the next container. JADE provides weak mobility since arbitrary execution state can not be transferred. Execution of the agent at the destination container resumes at a pre-determined point (the beginning of a behaviour) in the agent.

4 Core Interfaces of SAgent

SAgent uses sound principles of object-oriented design and achieves abstraction by specification of several general *interfaces*, which are extended by security providers to supply protocol-specific implementations. The design is similar to some well-designed cryptography libraries, such as Sun's Java Cryptography Architecture, in which generic "encryption" and "decryption" interfaces are defined, and these are bound to specific algorithms and implementations when the cryptography objects are allocated. The design of 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 which can travel with the agent. The core of our design, therefore, includes secure private and public interfaces, along

with a secure data interface. 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.

In this rest of this section we describe these core interfaces in detail, and describe the functionality they specify. The views and use of these interfaces from both the security provider and application developer are described in more detail in the following section. For reference in the following discussion, the private and public interfaces defined in SAgent are shown in Figure 2.

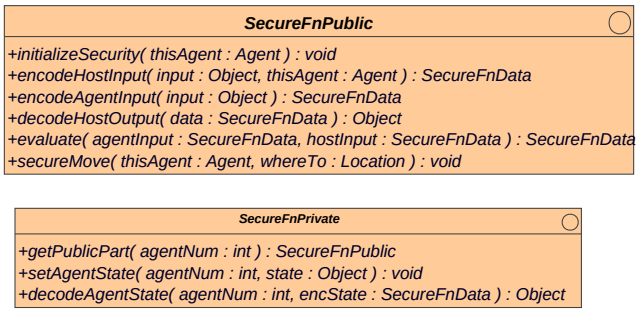

Figure 2: Core interfaces of SAgent

4.1 SecureFnPrivate Interface

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. The originator creates an agent for a particular application using a constructor given by a particular security provider, with the result being a `SecureFnPrivate` object. Through this interface, the originator sets the initial state of the agent, and retrieves the "public part" (a `SecureFnPublic` object) which is then sent out in the mobile agents for this application. Each mobile agent then visits various remote hosts, utilizing its public functionality to perform computations on behalf of the originator. Note that the agent can perform both unprotected and protected computations at remote hosts, and good design would suggest that only the sensitive parts of the agent application be performed within the SAgent framework. The `SecureFnPrivate` interface also defines the functionality that allows the originator to decode the state of the agent

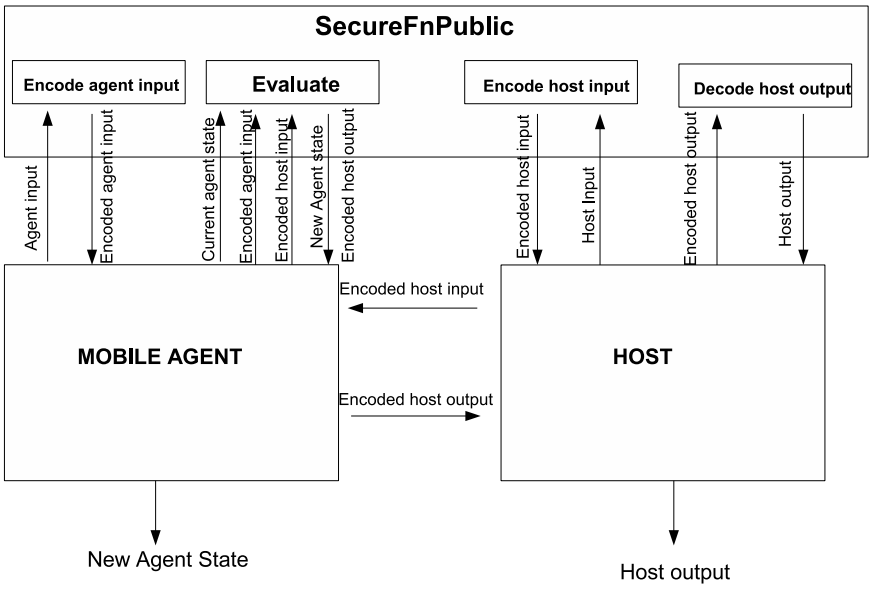

Figure 3: Component interactions in SAgent

after the agent returns home. After successful decryption of the agent state, the state of the agent is converted from a secure data format to a generic object, understood by the application. Thus, it can be seen that the `SecureFnPrivate` interface provides exactly those methods that allow the originator to perform operations that no remote host should be allowed to do.

4.2 SecureFnPublic Interface

The `SecureFnPublic` interface encapsulates the functionality required by the agent to perform protected computations at remote hosts, modeled as described earlier and shown in Figure 1. 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. The encoding is a conversion from a generic object into an opaque `SecureFnData` object, which involves a two-step conversion behind the scenes: the application-specific data is first converted into the intermediate, generic data format, and then the protocol-specific conversion of the intermediate data into a protected format is done. The `SecureFnPublic` interface also defines the function to evaluate the protected agent functionality on these inputs. Below, we describe how the public functionality of an agent 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 3 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 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, where security is defined by the particular protocol or application and can refer to integrity and/or privacy of the data.

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.

4.3 SecureFnData Interface

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. Therefore, the key idea with SAgent is that only *protected* data objects should be allowed within the protected execution. The `SecureFnData` type is simply a generic, opaque type to represent protected data, and does not provide any methods for operating on this data — all operations are performed using either `SecureFnPublic` or `SecureFnPrivate` methods, including conversion to/from insecure data representations from/to `SecureFnData` objects.

Different applications have different data requirements, so 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. The security provider just deals with bits which he encrypts, decrypts, and operates on as binary data, and is unaware of the actual form or meaning of the data in any application that may use his protocol. Once protected as a `SecureFnData` object, the actual encrypted representation of the data remains opaque to the user of the protocol. The application-developer, on the other hand, provides conversion routines between generic binary data and application-specific data. Figure 4

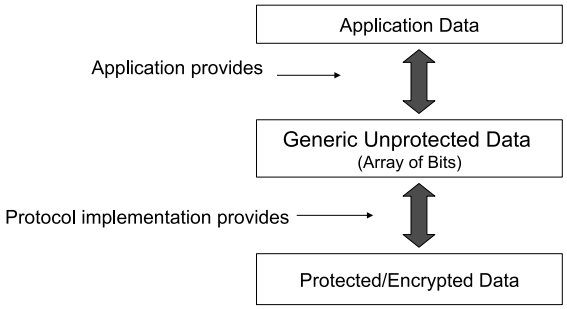

Figure 4: Representation of data in SAgent

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 the application developer 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.

4.4 HostAgent

In JADE, all entities of the mobile agent paradigm are agents, so a remote host in SAgent is an extension of the core SAgent class `HostAgent`, which is in turn an extension of the JADE class `Agent`. `HostAgent` encapsulates the functionality of a generic remote host in the mobile agent model. It includes functionality that allows the host container to register with the main controller of the JADE platform as an SAgent service provider. This registration process involves the host's specifying what SAgent application it supports and what protocols that it is willing to operate under, giving the host the ability to specify how it wishes its own data to be protected. For example, if a host supports the "Buyagent" application with the ACCK protocol, the service registered is "Buyagent/ACCK". If the host is unconcerned about which protection technique is used, it can register using special protocol name "any" (e.g., using "Buyagent/any"). This allows the integration of new protocols and applications seamlessly into SAgent without requiring any changes to the core SAgent architecture. This base class also has methods that facilitate exchange of objects between the agent and the host to allow the agent to perform computations at a visited host.

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. SAgent borrows the idea of "security providers" from the Java Cryptography Extension (JCE) [5] to allow programmers to independently develop their own implementations of various security protocols. 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. Since this is a common representation, classes used by both provided protocols are in package `sagent.util.circuit`, and are available for other protocol designs that use this technique. For software-based techniques using homomorphic encryption for smaller classes of functions, such as the technique of Sander and Tschudin [9], 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.

As pointed out above, new protection techniques based on using Yao's encrypted circuit idea are easy to experiment with, since the necessary circuit and signal implementations are supplied with SAgent.

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.* 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.

Step 4: *Create the mobile agents.* The protected part of a mobile agent is already determined by the public part passed along by the originator, so the protected functionality consists of some generic calls through the SAgent interface to retrieve the host input, encode it's own input, evaluate the function, and supply the host its output. Sample code from our sample 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 Experiments and Efficiency

In this section we present experimental results from running the applications described earlier under both protocols provided with SAgent. Experiments on the protocols themselves, showing the how complexity grows with input size and how parameters such as cryptographic algorithms and key sizes affect efficiency are being performed, but there are too many parameters and variations to present complete experimental results in this paper. Our purpose here is to provide some basic results that demonstrate the feasibility of our security framework and protocol implementations. The experiments were performed on a cluster of 7 Pentium IV, 2 GHz machines, all running Fedora Core 4 Linux with Sun's Java SDK 1.5 and JADE version 3.2. We designated one machine as the originator, and for each of the protocols used 3 agents visiting the 6 remote hosts (remaining 6 machines). For baseline measurements we used a simple protocol called *Insecure*, where the agents simply visited the hosts and performed the same operations but with unprotected techniques.

The running times from our experiments are shown in the following table, where times are cumulative times given in seconds.

Application	Protocol		
	Insecure	ACCK	TX
Maxbid	0.68	4.63	32.4
BuyAgent	0.74	15.1	140.0

Note that in all cases except the more involved BuyAgent application running under the TX protocol, the times are just a few seconds per host visited, which is quite a low overhead for the rigorous security guarantees that these protocols provide.

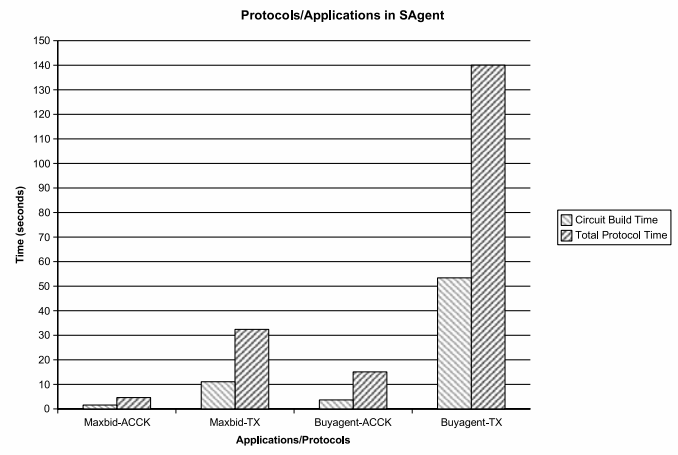

Figure 5: Comparison of ACCK and TX protocols in SAgent

Figure 5 shows the results graphically, with initialization and total time separated. The initialization time is the time for the originator to create the encrypted circuits for the applications. These results illustrate the trade-off between

efficiency and security for the protocols, with the TX protocol being less efficient than the ACCK protocol, while requiring fewer assumptions on trustworthiness of participants.

7 Conclusions

We have developed a comprehensive security framework called SAgent for the JADE multiagent platform. Our framework allows the development of secure protocols and mobile agent applications, and can be used to provide both confidentiality and integrity protections for agent data. Use of generic, application-independent interfaces and data formats within SAgent allows programmers to create applications and protocols with ease and transparency. We also provide the first software implementations of the ACCK [1] and the TX [10] mobile agent privacy protocols, with experimental results showing the feasibility of these software-only protections within our framework. We leave as future work the inclusion of support for hardware-based protections in SAgent, but believe that such an implementation would be straight-forward in a system that provides verifiable Java environments through hardware-based attestation. We also leave detailed experimental explorations of the two provided security protocols to a future publication.

8 Acknowledgments

SAgent has evolved from experimental security protocol testing code developed by Ke Xu as part of his PhD dissertation. While the interfaces described in this paper are all new, much of his code was retained in the protocol implementations and we are grateful for his initial work.

References

[1] J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth. Cryptographic security for mobile code. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 2–11, 2001.

[2] C. Bryce. A security framework for a mobile agent system. In *Proc. of the 6th European Symposium on Research in Computer Security*, volume 1895 of *Lecture Notes In Computer Science*, pages 273 – 290, 2000.

[3] J. Claessens, B. Preneel, and J. Vandewalle. (How) can mobile agents do secure electronic transactions on untrusted hosts? A survey of the security issues and the current solutions. *ACM Trans. Inter. Tech.*, 3(1):28–48, 2003.

[4] JADE, documentation and resources. available at: http://jade.tilab.com/.

[5] Java Cryptography Extension. http://sun.java.com/jce.

[6] Microsoft next-generation secure computing base — Technical FAQ. http://www.microsoft.com/technet/Security/news/ngscb.mspx, Feb. 2003.

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

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

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

[10] S. R. Tate and K. 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.

[11] Trusted Computing Group. Web site. http://www.trustedcomputinggroup.org.

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

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

[14] B. Yee. A sanctuary for mobile agents. *Secure Internet Programming, Lecture Notes in Computer Science*, 1603:261–273, 1999.

[15] X. Yi, C. K. Siew, X. F. Wang, and E. Okamoto. A secure agent-based framework for internet trading in mobile computing environments. *Journal of Distributed and Parallel Databases*, Volume 8, Number 1:85 – 117, January 2000 2000.

[16] M. Zhang, A. Karmouch, and R. Impey. Adding security features to fipa agent platforms. In *Mobile Agents for Telecommunicatoins Applications (MATA)*, 2001.