Exploring Data Integrity Protection in SAgent*

Vandana Gunupudi[1] and Stephen R. Tate[2]

[1] Dept. of Computer Science and Engineering, University of North Texas
gunupudi@cs.unt.edu
[2] Dept. of Computer Science and Engineering, University of North Texas srt@cs.unt.edu

Abstract. In this paper we show how agent data integrity protections can be provided in SAgent, which is a security framework designed for comprehensive protection of mobile agents in JADE. While SAgent was designed for comprehensive security protocols with different design and interaction criteria, we show how efficient integrity-only protections can be seamlessly integrated into SAgent. In situations where the more challenging problem of working with secret data on remote hosts is not needed, these integrity protection techniques provide significantly better performance than the comprehensive protections, as we demonstrate through a series of experiments.

1 Introduction

This paper presents the design of a new "security provider" that supports data integrity protection for SAgent [4], which is a general-purpose agent security framework for JADE [6]. Protection of the data carried by a mobile agent is a key issue in mobile agent security, and is concerned with protecting the confidentiality and/or integrity of data which is used in agent computations. SAgent was designed for comprehensive protection of confidentiality and integrity of agent data, with the ability to carry secret data that the agents can still use in computations while leaking no information about the protected data. While the comprehensive protections work well for protecting small functions, these powerful protocols do not scale well to more complex computations [2]. For this reason, it's important to consider the option of protecting just integrity of the data if working with confidential data isn't necessary. Note that when we refer to "integrity-only" protections, we are referring to data which can be *used* in computations — confidentiality of data which needs to be carried but not used until the agent returns to the originator is trivially protected simply by encrypting it with the originator's public key, so does not require the kinds of protections which SAgent provides.

In this paper, we provide details of the design and implementation of four *integrity* protocols in SAgent. These methods, namely, the Partial Result Authentication Codes (PRAC) [11], Hash Chaining [7], Set Authentication Code methods [9], and the Modified Set Authentication Code method [3], protect the integrity of data carried by mobile agents. This work builds upon the theoretical details of these methods proposed by various authors as well as our previous work in which we presented a performance evaluation of these methods, but which were implemented in an ad hoc manner on a very

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

basic mobile code platform [3]. By contrast, the current study uses the popular JADE platform, and provides a consistent and well-designed approach that allows agent application developers to easily take advantage of these protections.

Our results show that significant efficiency gains can be obtained with integrity-only protections, as compared with the more comprehensive protocols which were explored in depth in a separate paper [2]. In our previous work on integrity protections [3], we proposed a common notation to describe various phases of the data integrity methods. In this paper, we show how these distinct phases can be mapped seamlessly into SAgent. We conclude with a comparison of the overhead of protecting the confidentiality of agent data versus protecting the integrity of agent data as well as a discussion of the trade-offs between security and efficiency for the integrity methods.

2 Previous Work

The work described in this paper involves implementation of four techniques for protecting the integrity of agent data, which we briefly introduce in this section. For details, refer to the cited papers. While not all techniques require interaction, we generically refer to protections as "protocols" in order to have a common terminology. Methods that have been devised to protect the integrity of the data carried by the agent include the Partial Result Authentication Code [11], Hash Chaining [7], and the Set Authentication Code [8] methods. Yee proposed the Partial Result Authentication Codes method [11], whereby the result of the agent's computation at each host is encapsulated using a Message Authentication Code (MAC). The result at each host, combined with the corresponding MAC, is called the Partial Result Authentication Code, or PRAC. This method requires the agent to generate a secret key (used to calculate the MAC) for each host, using a one-way function, from an initial secret key given by the originator. This method ensures that none of the results collected prior to a malicious host can be modified without detection.

In the Hash Chaining [7] method, this idea is extended to chain the partial result to the identity of the next host to be visited. This method allows the originator to determine where exactly the chaining broke if a malicious host manipulates any of the partial results. This method, while providing stronger security, is not flexible enough to allow efficient updating of previous data provided by the hosts. New data can be added to the agent, effectively overriding earlier data, but the old data must be retained in order to maintain the integrity of the entire chain.

To address this problem and allow data updates, Loureiro, Molva, and Panetrat [9] devise an original cryptographic technique called a "Set Authentication Code," which allows the updating of an integrity proof which is initiated by the originator when the agent is created. Each host exchanges a secret key with the originator and uses this key to calculate the MAC on its results, and this MAC is used to update the integrity proof. The originator can efficiently verify this integrity proof upon the agent's return. Unlike hash chaining, a host can remove their own previous data and replace it with new information, effectively changing data that they have previously provided. Because this method allows secure updating of agent data, it is useful in dynamic scenarios like online auctions.

The Modified Set Authentication Code method was proposed in order to address some limitations of the Set Authentication Code method [3]. In particular, this technique modifies the way in which secret keys are managed, which involves using a key that can be integrated into the set of offers carried by the agent. The secret key is encrypted with the originator's public key (carried by the agent) and is then added to the set of offers.

This paper does not offer any new integrity protection protocols, but does provide a thorough study of the practicality of these protocols in the easy-to-use framework provided by SAgent. We consider both absolute performance and the trade-offs that can be made between security and efficiency for these methods.

3 Design

In this section, we describe the design of an "integrity" security provider for SAgent. We have produced implementations of four protocols that protect the integrity of agent data, and a secure mobile agent application that requires the protection provided by these protocols. We briefly describe the salient features of SAgent and then outline the design methodology used for the protocols and the application.

SAgent is a generic agent security framework that is designed to protect the computations of mobile agent applications. Analogous to the concept of distinct private and public keys in public key cryptography, SAgent separates the public and private portions of the mobile agent application into distinct pieces. 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. 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. Further, SAgent also distinguishes between a mobile agent *application* and a security *protocol* that protects specific parts of an agent application. Security protocols are normally defined without regard to the different types of applications that may use them. Similarly, applications are not necessarily designed for specific protocols; in fact, some applications may not even require security protection. This creates two different perspectives of SAgent — one of a programmer that develops protection techniques for SAgent and the other of a developer who writes applications for SAgent. The architecture of SAgent is designed so that the security provider and the application-developer can 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.

3.1 Design of integrity protocols in SAgent

Since SAgent separates the mobile agent computation into its private and public parts, we first identify the public and private functionality of each protocol. Because SAgent provides a clean abstraction of data protection in agents by defining a number of general security interfaces, each of our protocols implements the core private, public and data security interfaces of SAgent called `SecureFnPublic` and `SecureFnPublic` and `SecureFnData`, respectively. Next, the security provider must decide what the secure representation of data will be within SAgent. We define a secure data interface to represent integrity data within SAgent, which we call `IntegrityFnData`. The secure application, which we call `MinBid`, simply consists of a mobile agent that visits various remote hosts, collecting offers for the price of an item, allowing the hosts to update their offers after seeing the lowest offer, if they so desire. This simple application simulates an interactive bidding scenario like an online auction, whose participants can bid against each other for an item. Notice that in contrast to the `Maxbid` application in the standard SAgent distribution, the `MinBid` application allows the hosts to see each other's offers since we are interested in protecting the *integrity* but not the confidentiality of the agent data. The `MinBid` application deals with Integer bids and we provide routines to translate these bids into a generic form that can be used by the integrity protocols within SAgent.

We next describe the key components of the public and private interfaces of the protocols.

- **Initialization.** The initialization step involves the originator encoding the public functionality into the agent. The initialization process also ensures that any protocol-specific data required by the agent's public functionality are made available to the agent. The initialization step involves the originator calling the constructor of the appropriate `SecureFnPrivate` class, for example, `PRACFnPrivate` for the PRAC method.
- **Encoding of the Host input.** When the mobile agent reaches a host, it interacts with the host via the host agent. If the host offers a valid bid for the item in question, the agent encodes the host's offer in a protocol-appropriate manner. In this case, this step involves simply converting the host's Integer input into an IntegrityFnData object that can be operated upon within SAgent.
- **Evaluation of the input.** Once the agent has the encoded host input, it uses its public functionality to evaluate the input. The specifics of the evaluation vary from protocol to protocol but every integrity protocol involves the creation of an integrity proof for the host's offer. The agent state is updated as a result of this evaluation.
- **Movement of the agent.** Once the encapsulation process is completed, the agent migrates to the next host on its itinerary through the *secureMove* method. The next host on its itinerary is either another remote host or the originator.
- **Finalization.** When the agent returns home, it returns its state to the originator in the form of an `IntegrityFnData` object which the originator can decode using the `decodeAgentState` method. This completes the verification of the integrity of the offers collected by the agent.

3.2 Design of a secure Application

The secure `MinBid` application consists of several classes: a `MinBidOriginator`, a `MinBidMobileAgent` and the host agents. To design this application, we performed the following steps:

- **Decompose the application into its protected and unprotected portions**: This step involves deciding which parts of the application need to be protected. In the `MinBid` application, hosts are allowed to update their offer based on the current minimum bid carried by the agent, so our application keeps track of the minimum bid in the agent state. The `MinBid` application protects the integrity of each offer collected by the agent.
- **Create an originator agent**: The originator must call the constructor of the appropriate implementation of the `SecureFnPrivate` interface, passing the protected application part, i.e., `IntegrityMinBid`, created earlier. The originator now retrieves the public part of the application and creates mobile agents which can carry this public part. After creating the agents, the originator simply waits for the agent(s) to return after collecting the offers from various hosts.
- **Create the mobile agents**: The mobile agent interacts with the host, specified by a `HostAgent` which supplies its input to the agent. The mobile agent then uses its public functionality to perform computations on the data at the host, including protecting the computation in the appropriate manner. Since the `MinBid` application allows updates of previous offers, a mobile agent is allowed to visit a host more than once.
- **HostAgent**: Since all entities in JADE are agents, we create a HostAgent to represent the functionality of a remote host in the agent paradigm. The host agent in this application generates a random bid, based on a seed to a pseudo random number generator. The purpose of seeding the pseudo random number generator with the same seed each time is to ensure that the host supplies the same bid each time, so that we can measure the times consistently. The host simply generates a random bid, checks if it is greater than its threshold limit and, if so, offers a bid less than the current minimum.

4 Integrity Protocols

In section 2, we described the integrity protocols briefly. In this section, we present a more detailed overview of the protocols, identifying the distinct phases of each protocol and showing how each phase can be integrated seamlessly into the SAgent framework.

4.1 Security Model

In order to compare the data integrity methods, we use the example of a comparison-shopping agent, cited in mobile agent literature [7]. The agent owner (henceforth known as the originator) wishes to obtain prices from various servers (online shops) to buy a certain item. The originator programs the agent to visit different servers, querying them for the price of the item. The originator goes offline after sending the agent out, while

the agent visits each host on its itinerary, collecting prices from each host, and returns to the originator with the set of competing offers. In a competitive bidding scenario, the agent may visit each host more than once. The originator will buy the item from the host that offers the lowest price (or highest desirability), based on the agent's set of offers. Above all, the originator must be assured that the set of offers is valid, i.e., each offer in the set is an authentic bid from the particular host, and that no malicious host modified any competing offer. Each of these protocols has distinct phases: Initial setup by the originator, initial visit on intermediate hosts, updating of data, and verification of data integrity. We present a uniform notation developed in our previous work [3] to formally describe each step of the the protocols.

S_0	Originator
S_i $(i \geq 1)$	Hosts on the agent's itinerary (Intermediate hosts)
o_i	Actual offer from S_i
O_i	Encapsulated offer from S_i
$h(x)$	Cryptographic hash function
r_i	Nonce generated by S_i
$MAC_{k_i}(x)$	Message Authentication Code generated using secret key k_i

It should be noted that in this paper, we only provide details about the phases of the protocols only as they map into SAgent and make no statements about the security properties. For an extensive discussion of the security properties and features of these protocols, refer to [3]. The agent visits a sequence of hosts $S_1, S_2, \ldots, S_n$, and obtains an offer o_i from each host, where an offer includes not only a price but some indication of the source of the offer[3]. Notationally, we say the agent carries a set of offers ω, a set of encapsulated offers Ω, and a key k, and we use subscripts to denote these values over time. For example, ω_i represents the value of ω after the agent has visited the ith host.

4.2 Partial Result Authentication Codes

The Partial Result Authentication Code (PRAC) method involves the calculation of a MAC on the offer at each host, using a secret key. It can be easily extended to an interactive scenario in which hosts bid against each other. Secure updates are possible using PRACs by allowing the host to retain the key used to calculate the MAC on the previous offer. The key must be removed from the agent to preserve forward integrity, but it can be retained securely by the host. If a host wants to update an offer, it simply uses the previous key and replaces the offer in the data set and the set of encapsulated offers. Upon agent return, the originator re-computes the MACs after generating the secret keys for all the visited hosts. If the MACs match, the bid is accepted.

Initialization The initialization phase of this protocol involves the generation of a secret key for the first host to be visited by the agent. This computation corresponds to

[3] Note that while we are using a comparison shopping example and the corresponding terminology, o_i could actually represent arbitrary state information from an agent after visiting S_i in any application.

the private functionality of the agent and is implemented in the `PRACFnPrivate` class which implements the `SecureFnPrivate` interface of SAgent. The private functionality is known only to the originator and is used to generate the secret key for the first host. The agent state is a generic object of type `PRACData` and the initial state includes the key for the first visited host. The state is converted into a secure `IntegrityFnData` object by the *setAgentState* method of the `PRACFnPrivate` class and incorporated into the agent.

Visit to a host When the agent is at a remote host, it retrieves a bid from the host and calculates a proof of its integrity, which is simply a message authentication code (MAC) on the offer, using the given secret key. The agent then uses the current secret key to compute a secret key for the next host in the agent's itinerary, if it is a first visit at the host. If it has previously visited the host, the previous key, stored securely by the host is re-used. Since these computations must be performed by the agent using information from its public part, this phase corresponds to the public functionality of the agent and is implemented in the `PRACFnPublic` class which implements the `SecureFnPublic` interface. In particular, this phase corresponds to the *evaluate* method of the public interface. At each host, the state of the agent is updated as a result of the evaluation of the host input. It should be noted that this evaluation is performed only on protected `IntegrityFnData` objects.

Verification of integrity When the agent returns, the originator decodes the state of the agent, converting the protected state into an unprotected object. This phase corresponds to the private functionality of the agent and is implemented in the *decodeAgentState* method of the `PRACFnPrivate` class. The originator computes the secret keys for all the visited hosts and verifies the integrity of each bid by computing a MAC on the offer with the corresponding secret key. Only upon successful verification, is a bid accepted.

4.3 Hash Chaining

Karjoth *et al*. [7] extended Yee's concept to ensure *strong forward integrity*, whereby none of the offers in the agent data can be modified without detection by the originator. Each offer in the data set is *chained* to the next one using the identity of the next host to be visited by the agent as well as a random nonce, encrypted with the originator's public key. Upon agent return, for each visited host, the originator decrypts the random nonce and generates the secret key used for each offer. If the chaining relation fails at a particular point, all subsequent offers are rejected.

Initialization The initialization phase of this protocol[4] involves the generation of a secret key for the first host to be visited by the agent. The secret key is generated using a random nonce, a random token of the agent instance, and the identity of the next host to be visited by the agent. Similar to the PRAC method, this computation corresponds to the private functionality of the agent and is implemented in corresponding `HCFnPrivate` class. The private information (nonce and token of agent instance) is thus used to generate the secret key and is known only to the originator.

[4] abbreviated as HC

The agent state is an object of type `HCData` and the initial state includes the key for the first host as well as the originator's RSA public key for use by the agent at a visited host. The generic state is converted into a secure `IntegrityFnData` object and incorporated into the agent.

Visit to a host When the agent is at a remote host, it retrieves a bid from the host and calculates a proof of its integrity. The integrity proof, again, is a MAC, which is calculated on the offer, a random nonce generated at the host, and the identity of the next host to be visited, using the given secret key. The agent then uses the current secret key, the current offer and the nonce to compute a secret key for the next host in the agent's itinerary. The random nonce is then encrypted using the originator's RSA public key. Since this method does not allow secure replacement of previous offers, these computations must be performed during each visit. Thus, this phase corresponds to the public functionality of the agent and is implemented in the `HCFnPublic` class. This step corresponds to the *evaluate* method of the public interface. At each host, the state of the agent is updated as a result of the evaluation of the host input. As before, this evaluation is performed only on protected `IntegrityFnData` objects.

Verification of integrity When the agent returns, the originator decodes the state of the agent, converting the protected state into an unprotected object. This phase corresponds to the private functionality of the agent and is implemented in the *decodeAgentState* method of the `HCFnPrivate` class. The originator decrypts the random nonces for each host, computes the secret keys for all the visited hosts and verifies the integrity of each bid by computing a MAC on the offer and the nonce with the corresponding secret key. Verification by the originator must be performed sequentially, i.e., knowledge of the sequence of hosts visited is mandatory for verification. Successful verification of integrity involves verification of the chaining mechanism at each host for this method.

4.4 Set Authentication Codes

We present a brief description of the Set Authentication Code[5] method. For a detailed description of the cryptographic mechanisms involved in this method, refer to [8]. Each visited host $S_{i>0}$ (i.e., each host except the originator) exchanges a secret shared key k_i with source S_0, using the Diffie-Hellman key exchange technique. The originator, S_0, then sends the agent to visit a set of hosts $S_1, S_2, \ldots, S_n$ with an initial set integrity value Γ_0 and empty data collection list ω_0. The set authentication code function, $\Gamma()$, is maintained over the set of the MACs calculated on the offers (i.e., encapsulated offers) through the use of the $insertfn$ and $deletefn$ functions. This set is conceptually the same as Ω in the previous methods, but may not be carried with the agent in the set authentication code method. In this case, the single integrity proof value substitutes for the set Ω, and revealing individual encapsulated offers in Ω would allow hosts to modify the integrity proof to change bids.

Initialization The initialization phase of this protocol involves the generation of an overall integrity proof and the generation of parameters for the Diffie-Hellman key

[5] abbreviated as SAC

exchange. The agent state is an object of type `SHData` and the initial state includes the initial overall proof as well as the originator's Diffie-Hellman public parameters. The state is converted into a secure `IntegrityFnData` object and incorporated into the agent. This computation is performed by the originator and is implemented in `SHFnPrivate` class which implements the `SecureFnPrivate` interface of SAgent. The private functionality is thus used to generate the initial integrity proof and the parameters used for generating the proof are kept secret by the originator.

Visit to a host When the agent is at a remote host, it retrieves a bid from the host. It then uses the originator's public Diffie-Hellman parameters to complete the Diffie-Hellman key exchange and generate a secret key. It then calculates a message authentication code (MAC), using the given secret key on the offer. It then uses the MAC to update the overall integrity proof. The overall integrity proof is modified securely by replacing the previous offer if the agent re-visits a host. As before, these computations correspond to the *evaluate* method of the `SHFnPublic` class or the public interface. At each host, the state of the agent is updated as a result of the evaluation of the host input. The updated agent state includes the overall proof and the public key of the current host. Again, this evaluation is performed only on protected `IntegrityFnData` objects.

Verification of integrity When the agent returns, the originator decodes the state of the agent, converting the protected state into an unprotected object of type `SHData`. This phase corresponds to the private functionality of the agent and is implemented in the *decodeAgentState* method of the `SHFnPrivate` class. The originator computes the secret keys for all the visited hosts using the Diffie-Hellman parameters and the public keys of the hosts. It then verifies the overall integrity of the bids by recomputing the overall integrity proof. If these proofs match, the set of offers is validated.

4.5 Modified Set Authentication Codes

In the Modified Set Authentication Code method[6], each secret key is generated entirely on the visited host and is encapsulated into the agent data after being encrypted with the originator's public key. For updates of offers, a host can either re-use its previous secret key or can generate a new key for each visit, but the host must remember the last key used in order to "cancel" its previous offer. The remainder of the technique is similar to the Set Authentication Code method. The originator S_0 sends an agent to visit a set of hosts $S_1, S_2, \ldots, S_n$ with an initial set integrity value Γ and an empty data collection list ω. None of the offers in the data set can be modified by a malicious host. Since the secret key is known only to the host and is carried in encrypted form within the agent, a malicious host cannot retrieve the secret key.

Initialization The initialization phase of this protocol involves the generation of an overall integrity proof. This computation corresponds to the private functionally of the agent and is implemented in `MSHFnPrivate` class which implements the `SecureFnPrivate` interface of SAgent. The agent state is an object of type

[6] abbreviated as MSAC

`MSHData` and the initial state includes the initial overall proof as well as the originator's RSA public key. The state is converted into a secure `IntegrityFnData` object and incorporated into the agent. The private information used to generate the initial integrity proof and the corresponding private key are kept secret by the originator.

Visit to a host At the remote host, the agent retrieves a bid from the host. It generates a random secret key and encrypts it using the originator's public key. It then calculates a message authentication code (MAC), using the given secret key on the offer. As before the overall integrity proof is updated using this MAC. The overall integrity proof is modified securely by replacing the previous offer/MAC if the agent re-visits a host. As before, these computations correspond to the *evaluate* method of the `MSHFnPublic` class or the public interface. At each host, the state of the agent is updated as a result of the evaluation of the host input. The updated agent state includes the overall proof and the encrypted secret key used by the current host. Again, this evaluation is performed only on protected `IntegrityFnData` objects.

Verification of integrity When the agent returns, the originator decodes the state of the agent, converting the protected state into an unprotected object of type `MSHData`. This phase corresponds to the private functionality of the agent and is implemented in the *decodeAgentState* method of the `MSHFnPrivate` class. The originator decrypts the secret keys used by the visited hosts using its private key. It then verifies the overall integrity of the bids by recomputing the overall integrity proof. If these proofs match, the set of offers is validated.

5 Experiments

The experiments were performed on a cluster of seven 2GHz Pentium IV machines, all running Fedora Core 4 Linux. We used Sun's Java SDK 1.5 with JADE version 3.2 and an instrumented version of SAgent 0.9. One machine was designated as the originator and there were 6 visited hosts. For baseline measurements we used a simple SAgent provider called *Insecure*, where the agents simply visited the hosts and performed the required computations though no integrity protection mechanisms were applied. We varied the bidding mechanism such that each host was visited multiple times, resulting in trials in which the number of host visits was 17, 31, 46, 66, 124, 168, 214, 289, 320 and 589. For each combination of parameters, we performed each test seven times, discarded the min and max values, and averaged the remaining results. For all implementations of data integrity mechanisms, the HMAC-SHA1 function was used to calculate the MACs on each offer, generating a 160-bit output. 1024-bit keys were used for both the RSA [10] implementation and the Diffie-Hellman key exchange mechanism [1]. We used the basic cryptographic algorithms that are part of Sun's Java Cryptography extension (JCE), and the Cryptix[5] JCE's implementation of the RSA algorithm. We used a modified form of the Diffie-Hellman key exchange [3] which allows the originator to remain offline, while allowing hosts to still perform the Diffie-Hellman key exchange with the originator using the parameters for the exchange which are part of the agent state.

5.1 Metrics

This section outlines the metrics used for measuring the performance of the integrity protocols. These metrics include basic protocol times which correspond to the various phases of the protocols as well as the agent size. In general, agent size increases when protection mechanisms are applied, so it is interesting to measure the size overhead of these methods.

- **Initialization time**: The initialization step involves the originator creating the public functionality of the agents, setting the initial agent state, and then sending the agents out to visit the hosts. This step involves the generation of initial keys for all the protocols, with a key being generated for the first host to be visited in the PRAC and Hash Chaining methods and the generation of Diffie-Hellman parameters for the Set Authentication Code method and the generation of the originator's RSA public key for the Modified Set Authentication Code method.
- **Agent Computation time**: This is time the agent spends on remote hosts, measured from when the originator sends the agents out to the time when they return. On each host, the agent retrieves the host's bid and creates an integrity value on it and adds the offer to the set of encapsulated offers carried by the agent.
- **Per-Visit Computation time**: This is the agent computation time, divided by the number of host visits. This metric gives a measure of the amount of time an agent spends performing computations on each host that it visits. For the `MinBid` application, the major part of the agent computation which involves the generation of a key for the host, happens during the agent's first visit. In subsequent visits, the host merely replaces it previous offer and updates the integrity value on the offer.
- **Verification time**: This is the time taken by the originator to verify the integrity of all the offers collected by the agent. Depending on the method, this step involves generation of keys for all visited hosts (PRAC and Hash Chaining) or the verification of an overall integrity proof for the Set Authentication Code and Modified Set Authentication Code methods.
- **Total Protocol time**: This is the total time from start to finish for each of the protocols. This includes the initialization time, the verification time as well as the total agent computation time.
- **Agent Size**: We measure the initial size of the agent after the originator sets its state and then measure the final size after the agent returns home.

6 Results

In this section, we present the results of experiments with the integrity protocols. In general, there are two types of bidding applications in which protection of the integrity of mobile agent data is required: an interactive bidding scenario like an online auction where the originator allows hosts to update their previous offers and the other in which the originator does not require updates of previous offers. The interactive-bidding scenario has wider applicability, so we focus on this scenario when performing tests.

Total Protocol Times Figure 1 shows the total times for each of the methods, with precise values given in the following table. As expected, the increase in protocol time is proportional to the number of host visits.

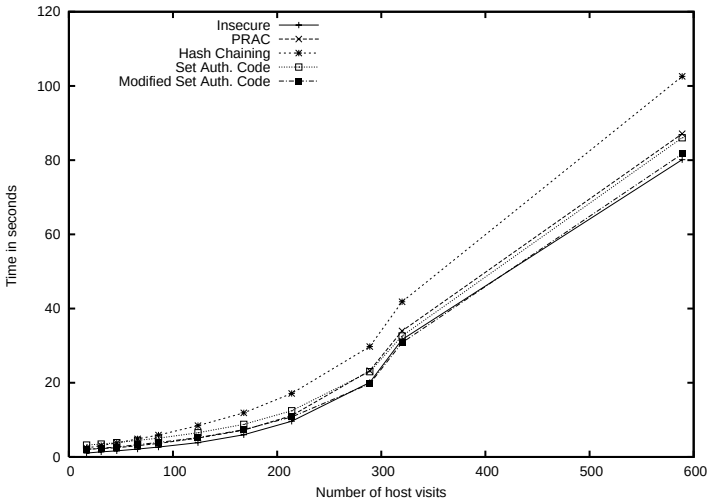

Fig. 1. Total Protocol time vs. Number of Host Visits

Number of host visits	Insecure	PRAC	HC	SAC	MSAC
17	1.05	2.06	2.395	3.22	2.04
31	1.37	2.26	3.07	3.49	2.38
46	1.65	2.49	3.78	3.85	2.68
66	2.12	3.04	4.89	4.47	3.31
86	2.68	3.64	5.90	5.06	3.91
124	3.87	5.10	8.43	6.53	5.19
168	5.98	7.22	11.88	8.78	7.36
214	9.62	11.13	17.12	12.45	10.79
289	20.08	23.26	29.76	23.0	19.74
320	31.67	33.97	41.85	32.61	30.82
589	80.14	87.12	102.60	86.05	81.67

We notice that the hash chaining method is the least efficient. Because this method does not allow updates in a secure manner, the agent must carry all previous bids given by a host to validate the integrity of the offer set, leading to higher overall times. The Set Authentication Code and the Modified Set Authentication Code methods, which use public key cryptography (and allow secure updates of previous offers without having to keep track of all previous offers), are quite efficient, requiring just over a minute to visit the hosts 600 times, with per host computation times of less than 0.2 seconds. They are comparable in efficiency to the simpler PRAC method, which requires the agent to carry the secret keys as part of its state. The time overhead due to the integrity mechanisms is very low for all the methods except the Hash Chaining method.

Next, we break the time down into its various components, and consider the size of the agents. The results are summarized in the following table.

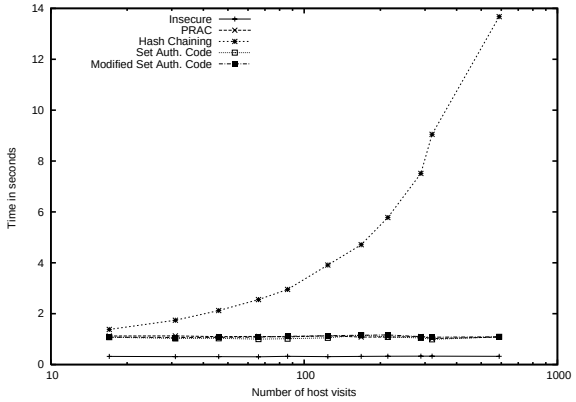

Fig. 2. Integrity Verification Time vs. Number of Host Visits

	Insecure	PRAC	HC	SAC	MSAC
Initialization time	0.03	0.04	0.20	0.37	0.21
Verification time	0.32	1.09	5.04	1.05	1.10
Final Size (bytes)	41939	44910	80348	47237	44315

Initialization times These average times for the initialization phase do not vary with the number of host visits, since the initialization step is independent of the number of host visits. This step merely consists of the originator setting the initial agent state and creating the agents. We perform this step offline, so the initialization time for the protocols includes the time for the originator to create the public portions of the agent and set the initial state of the agent.

Verification Times The verification times are graphed in Figure 2. The verification time is proportional to the number of offers collected by the agent. The verification times for the PRAC, Set Authentication Code, and the Modified Set Authentication Code methods are comparable and quite low, around 1 second on average, to verify the integrity of the set of six offers collected by the agent. For the Hash Chaining method, however, the verification time is much higher, proportional to the number of host visits, since the method does not allow secure replacement of previous offers. In order to verify the integrity of the offers in the Hash Chaining method, the originator must calculate the secret keys(which involves an expensive RSA decryption operation) for each host, and then compute a MAC on each offer, leading to higher verification times.

Agent Size Figure 3 graphs the initial size of the agent for the various protocols. The initial size does not show much variation across the protocols and is around 5000 bytes, on average. The final size of the agent is shown in figure 4, and, as expected, the agent size increases as the number of host visits increases. However, for all methods except the Hash Chaining method, the increase in agent size is not very dramatic since these methods allow secure replacement of previous offers. For the Hash Chaining method, the agent size increases rapidly with increasing number of

host visits, making it less attractive for interactive-bidding scenarios. For all other methods, the agent size is a relatively minor overhead.

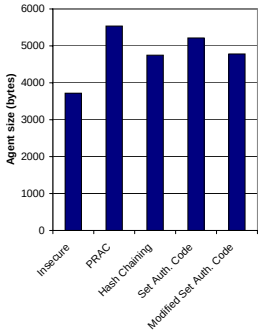

Fig. 3. Initial Size of the Agent

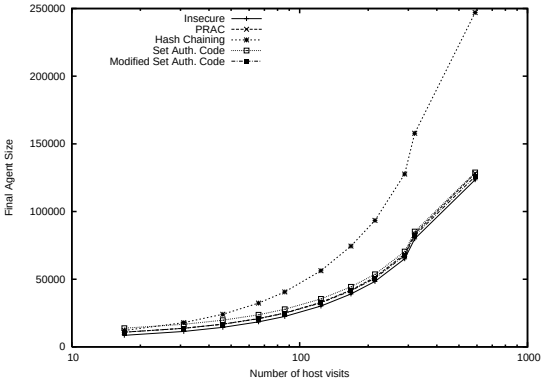

Fig. 4. Final Size of the Agent

Confidentiality versus Integrity We look at per-host times for the confidentiality protocols and corresponding per-visit computation times for the integrity protocols.

ACCK-1	TX-ECC-4	Insecure	PRAC	HC	SAC	MSAC
0.9	2.68	0.06	0.12	0.13	0.19	0.11

For the confidentiality methods, this time is the total agent computation time averaged over the number of visited hosts. For the integrity methods, the key metric is the number of host visits, i.e., the number of times an agent visits a remote host in order to update the offers. Notice that times for the confidentiality methods are much higher than those for the integrity methods. The faster confidentiality protocol (ACCK), is about 4.7 times slower than the slowest integrity protocol. We pay an even higher penalty for the distributed TX protocol. In general, there is a strong trade-off between security and efficiency when applying protection mechanisms to agent data and this trade-off is more pronounced for the confidentiality protocols. Users should consider carefully when to apply protection mechanisms to agent data and if confidentiality is not a strong requirement, integrity protections can be used.

From the experimental results, we see that the Modified Set Authentication Code method is the most efficient in the interactive bidding scenario. It also provide strong security guarantees and is recommended for use when integrity protection is required. The PRAC method, though quite efficient, has the serious limitation that the secret key is part of the agent state. Therefore, when other efficient alternatives are available, the use of this method is not recommended from a security standpoint. The Hash Chaining method, though providing strong security protection due to its chaining mechanism, is less efficient than the other methods. The increase in overall time and agent size becomes prohibitive as the number of host visits increases. The Set Authentication Code method is also quite efficient and provides much of the same security guarantees that the

Modified Set Authentication method offers, but is vulnerable to the man-in-the-middle attack due to the use of the Diffie-Hellman key exchange mechanism.

Summarizing, if integrity protection is required for an agent application, the Modified Set Authentication Code method is the best alternative, both from an efficiency and security standpoint.

7 Conclusion

In this paper, we presented the design of an integrity "security provider" for SAgent. Using the clean abstractions of SAgent, we can securely separate the mobile agent computation into its private and public parts and integrate the various phases of the integrity protocols into SAgent. This illustrates the generic applicability of the SAgent framework, which allows the implementation of secure protocols for providing secure agent solutions. We present results comparing the overhead incurred by the integrity mechanisms and compare the trade-offs between security and efficiency for the various methods. In general, the integrity mechanism are all quite efficient, with the Modified Set Authentication Code method providing the best overall combination of flexibility, security, and efficiency. When we compare the performance overhead for protecting the integrity of agent data versus the confidentiality of data, we notice that it is 5–25 times more expensive to protect confidentiality than just integrity. In conclusion, our results show that it is feasible to securely protect the integrity of agent data in real-world scenarios like online auctions, even if applications are too large or complex to use the more comprehensive security protocols.

References

1. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.
2. V. Gunupudi, S. Tate, and K. Xu. Experimental evaluation of security protocols in SAgent. Under review — available as CoPS Lab Technical Report 2006-01.
3. V. Gunupudi and S. R. Tate. Performance evaluation of data integrity mechanisms for mobile agents. In *Proceedings of 2004 IEEE Conference on Information Technology: Coding and Computing (ITCC)*, pages 62–69, 2004.
4. V. Gunupudi and S. R. Tate. SAgent: A security framework for JADE. In *Proceedings of AAMAS '06*, 2006.
5. Cryptix, documentation and resources. available at: http://www.cryptix.org/.
6. JADE, documentation and resources. available at: http://jade.tilab.com/.
7. G. Karjoth, N. Asokan, and C. Gülcü. Protecting the computation results of free-roaming agents. In *2nd International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 195–207. Springer-Verlag, 1998.
8. S. Loureiro. *Mobile Code Protection*. PhD thesis, ENST Paris / Institut Eurecom, 2001.
9. S. Loureiro, R. Molva, and A. Pannetrat. Secure data collection with updates. *Electronic Commerce Research Journal*, 1(1/2):119–130, 2001.
10. R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
11. B. Yee. A sanctuary for mobile agents. In J. Vitek and C. Jensen, editors, *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 261–273. Springer-Verlag, 1999.