Chapter 1

Introduction

1.1 Mobile Agent

With the extensive penetration of Internet technology in our everyday life, many new opportunities arise, especially in the field of commerce. E-Commerce, or electronic commerce, is born along with the Internet. The setting up of a virtual shop leads to an immediate presence in the electronic world for a merchant. Billions around the world will be able to view the products/services online, and purchase online. With this 'click-and-mortar' concept, there is no need for the rental of expensive shops in a prime location, nor the need for hiring sales promoters. All that is needed is a web presence on the Internet.

As technology evolves, there is more than to setting up a virtual shop in the Internet. With millions of virtual shops springing up in various parts of the world, it is impossible for a customer to manually browse through all the possible shops before making a purchase decision. The task is even more challenging when the pricing information is dependent on the customer's requirement and subjected to negotiation. No customer is able to explain and negotiate the requirements to various online shops and gathering pricing information one by one. There is an urgent need for automated information gathering and negotiation. To address this concern, mobile agent technology is coming into the limelight [14]. Mobile agent can be defined as a system situated within a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future [20]. A mobile agent will be able to automate certain tasks that were processed manually and make certain decisions intelligently with/without the interference of its owner. With this approach, information gathering can be performed automatically within the split of a second, negotiation of certain complexity can take place without the active participation of agent owners, and the decision making process can be more efficient and reliable.

One hindrance to the widespread adoption of mobile agent technology is the lack of security.

Let’s take a look at the following example: A customer Alice browses through Virtual CD Mall -- an online CD shop – looking for her favorite artist Mariah Carey’s latest album. Once she makes her selection, she clicks on the ‘Check out’ button. On the order confirmation page, she is asked to key in her VISA credit card number and expiry date. A few days later, the CD is mailed to Alice via registered mail. At the end of the month, Alice receives her VISA statement in which the online transaction is recorded correctly. She happily makes the payment and thinks everything is over.

A few months later, when the next VISA statement arrives, Alice is astonished to find her card was used to pay for Playboy, an infamous online adult entertainment provider.

She immediately contacts VISA and reports the incident. However, the investigation is unable to reveal anything. At the end, Alice and VISA has to share the loss.

When it comes to online transactions, security becomes the primary concern. Internet was developed without security in mind. Information is flowing from hubs to hubs before it reaches the destination. By simply tapping to wires or hubs, one can easily monitor all information traffic. In the above example, one possibility is as follows: Henry, a hacker that happens to be monitoring Internet traffic to Visual CD Mall when Alice makes the online purchase. When Alice keys in her VISA card number and expiry date, the unencrypted information is eavesdropped. Subsequently, Henry uses the information for his own online purchases.

When a mobile agent carries sensitive information and private mission to execute in a remote location, the agent owner must be assured of various issues so that the agent will not be compromised, the information carried by the agent won’t be stolen, the cash credit carried by the agent wont’ be misused, etc. Security will be the issue that has to be addressed carefully if mobile agent is to be used in the field of electronic commerce.

1.2 Security

Before going into the security mechanisms implemented under SAFER (Secure Agent Fabrication, Evolution , and Roaming) [1], it is necessary to describe general concerns about security. There are five general security concerns [29], namely, identification, authentication, secrecy, message integrity and non-repudiation.

Identification is the process of establishing the identity of a party. This may not sound difficult in the context of real world, but in digital world, identification is not so straightforward. For example, in the real life, if Alice is a frequent shopper of CD Mall, the assistant will be able to establish the identity of Alice the moment she walks into the mall. Alternatively, she can produce her frequent shopper card to establish her identity. In digital world, merchants and shoppers do not get to see each other in person, hence, mechanisms must be provided to establish the identity of each other. Depending on the level of security needed, different approaches can be taken. They vary from simply providing the frequent buyer number to producing a third-party issued digital certificate.

Authentication refers to the process in which the identity of a party is verified. If a shop assistant recognizes Alice by her face, identification and authentication takes place at the same time (assuming Alice does not have a twin sister). Or if the shop assistant does not know Alice by person, the matching of Alice's face and the one on the frequent shopper card produced by Alice verifies that the cardholder is indeed Alice. Unfortunately, over the Internet, there is no way to authenticate each other via the above means. If Alice identifies herself by frequent shopper number, then authentication can be done by requesting her to key in the password associated with her number. However, this only provides a certain level of authentication. If Henry manages to intercept Alice's password, he will be able to impersonate Alice to Virtual CD Mall. The more secure means is through the use of digital certificate. To verify that the holder of the certificate is indeed the owner of the certificate, a random challenge can be sent to the certificate holder and a signature on the challenge is returned as an authentication message. Although the principle sounds secure, a secure protocol is required to achieve it without subjecting the protocol to known security attacks.

Secrecy, or privacy, refers to the confidentiality of the information exchanged. In real life, important documents are locked or sealed to keep away prying eyes. Similarly, in digital world, various encryption techniques are available to digitally 'lock' information. Only the person with the secret key is able to 'unlock' the information. In the example previously, Alice's credit card number is lost because sensitive information is not encrypted during its transmission in open network. In the design of SAFER, a combination of symmetric key encryption and public key encryption is used to ensure secrecy.

Message integrity aims to ensure the message is transmitted without being tampered with or replaced. When Alice places a digital order to purchase the CD titled Honey by Mariah Carey, Henry may intercept the order, change the purchase item to History by Michael Jackson and sends the modified order to Virtual CD Mall. If Virtual CD Mall is unaware that the order has been tampered with, it will go ahead to ship the wrong CD to Alice. In cryptography, message integrity is ensured using a combination of public key encryption and one-way hash function.

Non-repudiation is probably one among the last remaining issues not addressed by e-commerce. It refers to the inability of any party to deny that the communication has taken place. For example, after placing an order with Virtual CD Mall, Alice should not be able to deny it and refuse to pay when the bill arrives. Similarly, Virtual CD Mall should not be able to deny that Alice has made a purchase for Honey in the event that it runs out of stock. The design of SAFER ensures that important events (such as agent roaming, contract signing, etc) are non-deniable.

1.3 SAFER – Secure Agent Fabrication Evolution and Roaming

SAFER, or Secure Agent Fabrication, Evolution and Roaming, is a mobile agent framework that is specially designed for the purpose of electronic commerce [1]. As the name implies, SAFER provides an execution framework for mobile agents that facilitates agent fabrication, agent evolution and agent roaming. Most important of all, security has been a prime concern from the first day of research [2-4] given the nature of E-Commerce. By building strong and efficient security mechanisms around itself, SAFER aims to provide a trustworthy framework for mobile agents, increasing trust factors to end users by providing the ability to trust, predictable performance and communication channel [17].

The overview of SAFER is shown in Figure 1. While SAFER is designed to establish various SAFER communities with various functional entities as shown in Figure 2, it can be shown in Figure 1 that in addition to the various design considerations above, SAFER is also designed to be an open framework to interface to non-SAFER communities.

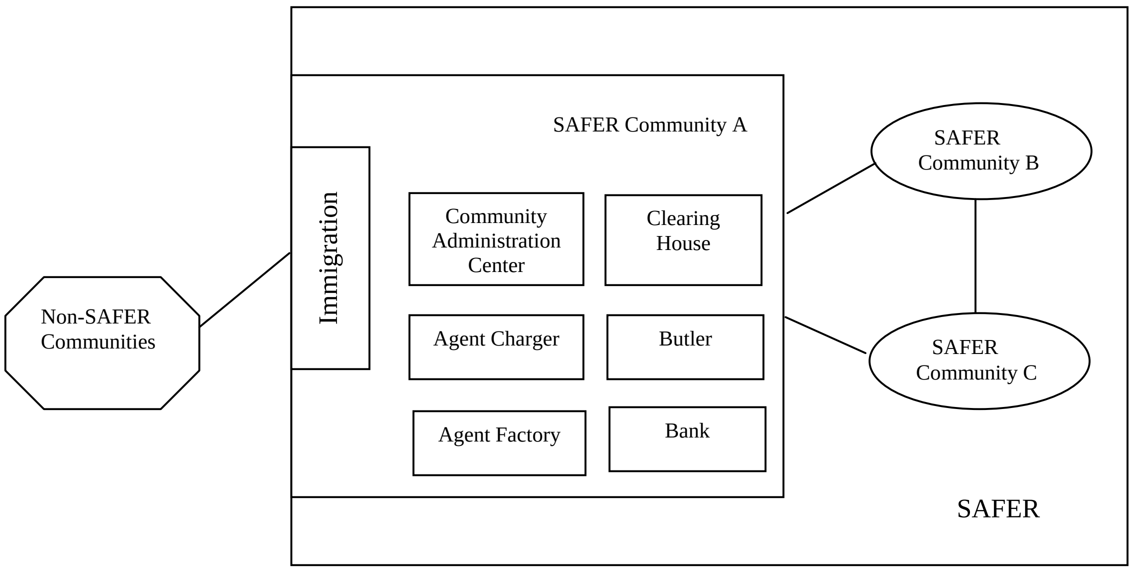


Figure 1 SAFER overview

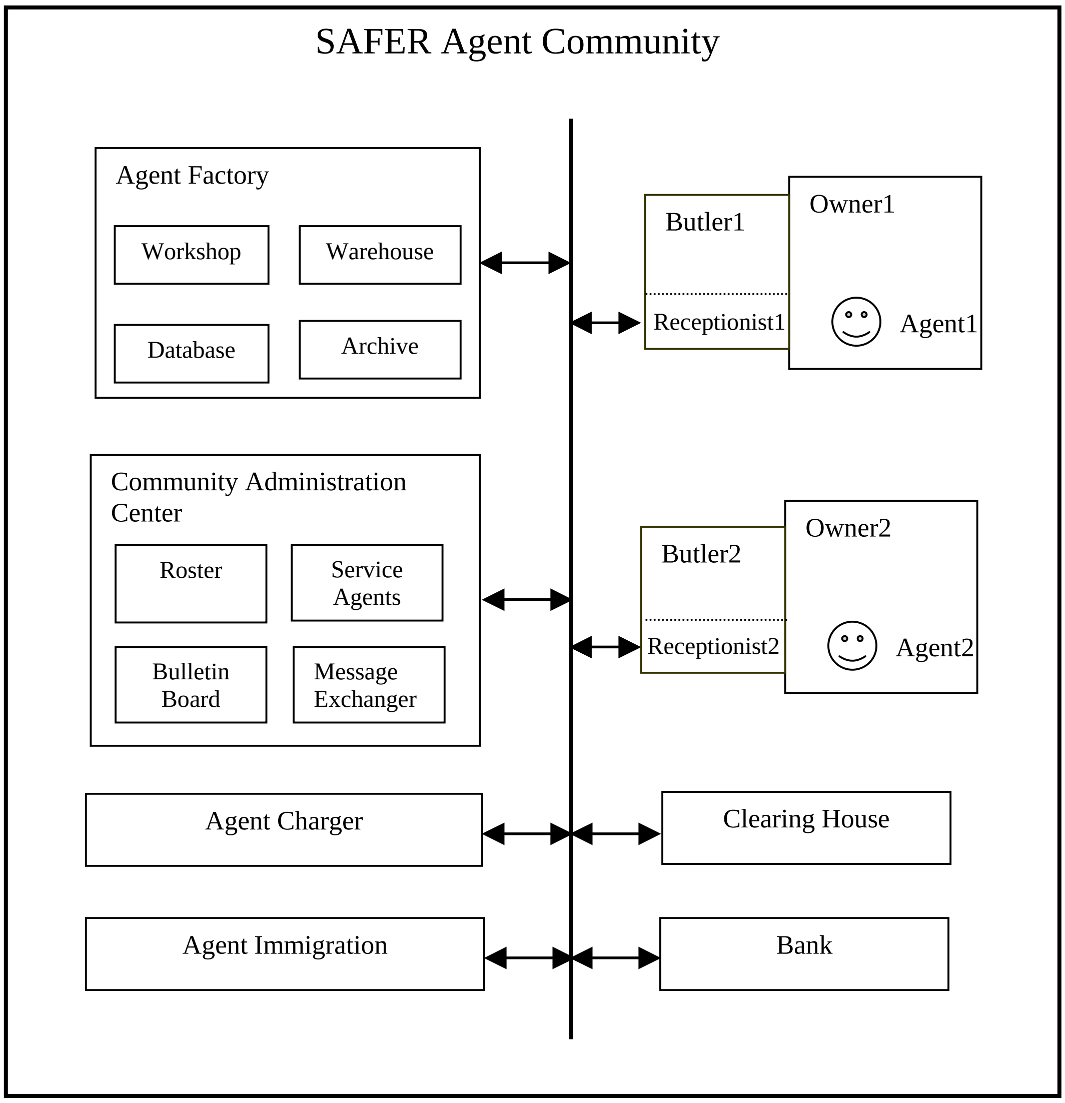


Figure 2 SAFER community

1.4 SADIS - SAFER Agent Data Integrity Shield

In the first paper on SAFER, a secure agent transport protocol is proposed to ensure roaming security (the secure agent transport protocol will be covered in subsequent section). While agent transport protocol provides for the secure roaming of agents, there are other areas related to security to be addressed.

Agent integrity is one such area that is crucial to the success of agent technology. Agent integrity refers to both agent code integrity and agent data integrity. Agent code can be defined as the executable module carried by an agent and any static information burned in the agent before it starts roaming. Agent code should not be modified under any circumstances during the agent's roaming session, thus the protection to its integrity can be achieved via conventional digital signature method. More complex code integrity scheme is also proposed in [6]. Different from agent code, agent data is dynamic in nature and will change as the agent roams from host to host. Despite the various attempts in the literature, there is no satisfactory solution to the problem so far. Some of the common weaknesses of the current schemes are vulnerabilities to revisit attack and illegal modification (including deletion/insertion) of agent data. AMP [5], an earlier proposal under SAFER to address agent data integrity, does address some of the weaknesses in the current literature. Unfortunately, the extensive use of PKI technology introduces too much overhead to the protocol. Also, AMP requires the agent to deposit its data collected to the butler before it roams to another host. While this is a viable and secure approach, our approach will provide an alternative by allowing the agent to carry the data by itself without depositing it (or the data hash) onto the butler.

Besides addressing the common vulnerabilities of current literature (revisit attack and data modification attack), SADIS also strives to achieve maximum efficiency without compromising security. It minimizes the use of PKI technology and relies on symmetric key encryption as much as possible, thus reducing the overhead introduced by the security mechanism to a minimum. In terms of data efficiency, it does not require the agent to carry any encryption key or random (for encryption key derivation) with it - some existing mechanism does require that. Instead, the data encryption key and the communication session key are both derivable from a key seed that is unique to the agent's roaming session in the current host. As a result, the butler can derive the communication session key and data encryption key directly.

Another feature in SADIS is strong security. The key seed negotiation itself is based on a variation of DH key exchange. During the negotiation, it also achieves the objective of implicit destination host authentication, and prevents the current host from getting any insight into the next key seed. Furthermore, to protect the key seed, it is never used directly as encryption key throughout the scheme. Instead, it is used to derive each session key and one-time data encryption key. Effectively, each message exchange between the agent butler and the agent is protected using a different session key. There is no correlation between each session key, making it extremely difficult for any attack on the keys.

Most of the existing research focused on how to detect integrity compromise, but neglected the need to identify the malicious host. With SADIS, the agent butler will not only be able to detect any compromise to data integrity, but to positively identify the malicious host.

1.5 SAT – Secure Agent Transport

A fundamental factor for agent roaming is the ability of mobile agents to roam from one host to another. Without roaming capability, the concept of mobile agent is no longer meaningful.

When a mobile agent roams from one host to another, a number of security concerns arise. Firstly, a number of general security concerns discussed above is applicable. In terms of data secrecy, information carried by the agent may be stolen during roaming, the code/execution logic carried by the agent may also be stolen. In terms of data integrity, the information on the agent may be modified by a malicious party, and the execution logic may also be modified. Another security concern is non-repudiation. There must be mechanisms available to ensure that the source host cannot deny that the agent has left for its destination afterwards, and similarly, that the destination host cannot deny that the agent has arrived. In addition, there are security concerns specific to agent roaming as well. For example, agents during roaming is vulnerable to abduction by malicious host, or ‘cloned’ maliciously by another host.

With all the security concerns mentioned above, agent transport becomes an important area to be addressed before establishing a SAFER framework. SAT, or Secure Agent Transport, is the agent roaming mechanism designed for SAFER to address all the above security concerns.

1.6 Approaches and Results

As the thesis focuses mainly on security related issues, the research focus is mostly motivated by the limitations of existing literature, especially flaws from existing protocols and mechanisms. The research problem is defined based on a literature review on current and updated literature. The two main problems addressed in this thesis include agent data integrity protection and agent transport security.

Before dwelling into the details of research, a list of assumptions is clearly stated and used as a basis for the research. The problem definition combines with the assumption statement essentially draws the boundary of the research focus and create a basis for subsequent research activities. Since the research is a part of the SAFER research initiatives, most of the assumptions stated are either assumptions from SAFER framework or issues already addressed/to be addressed by SAFER framework.

Unlike some of the previous security related researches, security researches in SAFER emphasizes on both security strength as well as efficiency. It is true that security always comes with a cost in efficiency. The research is conducted in such a way that defines security as a baseline and works around this security baseline to improve efficiency as much as possible. In this way, SAFER aims to achieve optimal efficiency while ensuring the security of the protocols. This principle is reflected in the design of SADIS and SAT in subsequent chapters. To further improve the applicability of the protocols, flexibility is also provided with the design. A number of different options are provided for the applications to choose based on the application's individual requirements. Based on the criticality of the mission and sensitivity of the information carried by the agent, the agent owner can choose the most appropriate protocol to satisfy the necessary security requirement and achieve higher productivity.

To verify the strength of the protocols designed, a comprehensive security analysis is performed on the protocol. Various attack scenarios are explored and analyzed. The strength of the protocol can be clearly illustrated in the security analysis as all foreseeable attacks can be protected against by the security mechanisms.

Prototyping is adopted as a general approach to evaluate the feasibility and test the efficiency of the protocols designed. Prototypes are developed for both SADIS and SAT. The results of the prototyping demonstrates the highly practicality of the protocols designed.

1.7 Structure of This Thesis

This thesis is organized into seven chapters. In this chapter, the overall SAFER framework and the motivations of the researches are introduced. The methodologies adopted for the researches are also described.

In Chapter 2, a comprehensive literature review is presented. The latest relevant literatures are analyzed and compared to the current research in this thesis. In the literature review, the motivation of the current research will be reinforced once again. At the same time, the contribution of current research can be seen through the various comparisons to the current research.

Chapter 3 is an overview of SADIS, SAFER Agent Data Integrity Shield. It presents an overall picture of SADIS protocol.

The details of SADIS are presented in Chapter 4. This includes the details of Key Seed Negotiation Protocol, Data Integrity Protection Protocol, security analysis of both protocols, implementation details as well as a section on the security attack simulation.

The overview of SAT, Secure Agent Transport, is presented in Chapter 5.

Detailed designs of SAT will be included in Chapter 6. The three different agent transport protocols, their security analysis and the result of the implementation will be covered.

Chapter 7 concludes the research on SADIS and SAT.

Chapter 2

Related Work

2.1 Related Work on Agent Integrity Protection (SADIS)

Agent data integrity has been a topic of active research in the literature for a while. There are various techniques to protect agent integrities [16], some of them based on trusted hardware, some of them based on trusted host, and some even based on conventional contractual agreements. In comparison, SADIS addresses the problem of data integrity protection via a combination of techniques discussed in [16]: execution tracing, encrypted payload, environmental key generation and undetacheable signature.

Over the years, there have been a lot of research targeted for agent integrity protection in one way or another. One of the newest active researches is the security architecture by Borselius [18]. The security architecture by Borselius aims at defining a complete security architecture designed for mobile agent systems. It categorizes security services into the following: agent management and control, agent communications service, agent security service, agent mobility service, and agent logging service. SADIS addresses the agent communication service as well as agent security services (integrity protection), while the research of SAT addresses agent mobility service.

While many of the security services are still under active research, the security mechanisms for protecting agents against malicious host is described in [19]. The paper proposes two mechanisms to protect mobile agents. The first mechanism makes use of a threshold scheme to protect mobile agents. Under the mechanism, a group of agents is dispatched to carry out the task, each agent carrying a vote. The agent votes for the best bid (under a trading scenario) independently. If more than n out of m ($m > n$) agents vote for the transaction, the agent owner will agree to the transaction. There are two modes of agent execution, one is to allow the agent to contact each merchant independently and gathers bid based on the given criteria, and the other to let the agent contact a subset of the merchants and communicates the best bid to its peers.

The first mode of execution effectively simplifies the agent roaming by allowing one agent visit one merchant only. While the approach avoids the potential danger of having the agent compromised by the subsequent host, it does not employ a mechanism to protect the agent against the current host. In the second mode of execution, the agent is required to communicate the best bid out of a subset of merchants to its peers. Firstly, specific security mechanism to protect the agent against subsequent merchants is not explicitly mentioned in [19]. Furthermore, if the agent's peer accepts the information without verification, the purpose of voting is defeated as the agent that collects the information can effectively manipulate the votes of its peers.

Most important of all, the threshold mechanism's security is based on the probability that no more than n hosts out of m are malicious. In another word, the security is established based on probability. Different from this approach, SADIS's security is completely based on its own merit without making any assumption about the integrity of external hosts. This is because the author believes that in an E-Commerce environment, security should not have any dependency on probability.

The second mechanism in the paper makes use of a trusted host to ‘supervise’ and ‘manage’ its agents. This is similar to the agent butler concept in SAFER. However, the mechanism requires each host to have a shared secret pre-established with the trusted host before the agents can interact with the host. Effectively, the trusted host must ‘know’ all the merchant hosts before sending out the agents. However, in the context of Internet, this is quite impossible and may significantly reduce the roaming scope of the agents. In SADIS, there is no requirement for such pre-established secrets between any entities. This allows any agent in SADIS to visit any hosts in the community, regardless whether the agent ‘knows’ the destination prior to the visit or not. This feature ensures that SADIS is more suitable for the practical E-Commerce environment.

While the research in [18] [19] is actively underway, there are other more mature researches in the area. One of such research works on agent protection is SOMA.

SOMA [11], or Secure and Open Mobile Agent, developed by University of Bologna, is a Java-based mobile agent framework that provides for scalability, openness, security in the Internet. One of the research focuses of SOMA is to protect mobile agent’s data integrity. To achieve this, SOMA makes use of two mechanisms: Multi Hop (MH) Protocol and Trusted Third Party (TTP) Protocol.

The advantage of MH protocol is that it does not require any trusted third party or even the agent butler for its operation. This is a highly desirable feature for agent integrity protection protocol. Unfortunately, MH protocol does not hold well against revisit attack when the agent visits two or more collaborating malicious hosts during one roaming session [5]. This limitation indicates that the security of MH protocol is based on probability (that the agent does not visit two or more collaborating malicious hosts). If the agent visits host *n* and host *m* ($n < m$) who happen to be both malicious and collaborating, there are a number of attacks possible. Firstly, host *m* can effectively wipe out the roaming record between *n* and *m* completely, producing an illusion to the butler that the agent hops from host *n* to host *m* directly. Another possible attack is for host *n* to 'make provisions' for data insertion before dispatching the agent for the next host. At any time before the agent returns home, host *n* can insert data into the agent using the provisions it made earlier. Lastly, host *n* can even modify the data it provides to the agent when the agent reaches host *m*. More detailed descriptions to the attacks on MH protocol was provided in [7].

Trusted Third Party protocol uses a different approach towards agent integrity protection. Sensitive operations (e.g. data hash calculation) are performed within a trusted environment so that the result can be certified and fully trusted. While this is definitely a secure mechanism, it does introduce significant overhead and inconvenience to the infrastructure. Since the agent has to visit a trusted host every time it needs to perform such sensitive operations, trusted third party host must be deployed in the network to facilitate such operations. Putting aside the practical administration and logistics issue, the introduction of these hosts opens up more points of failure and attack for hackers. If one of the many TTPs is broken into, all agents that visit this TTP may be compromised.

As a result, TTP and MH are used in combination to provide optimal security and efficiency under SOMA. However, given the nature of MH and TTP protocols, the security of its combined use is still subjected to the probability that the agent does not visit two collaborating host between visit to TTPs. In this thesis, we will propose a solution that does not base its security on probability.

Another agent system that addresses data integrity is Ajanta [8]. Ajanta is a platform for agent-based application on the Internet developed in the University of Minnesota. It makes use of an append-only container for agent data integrity protection. The main objective is to allow host to append new data to the container but prevents anyone from modifying the previous data without being detected. To achieve the objective, a checksum is calculated based on the previous checksum and the signed data from a new host. All the checksums are kept in the container for verification purpose later.

Similar to the MH protocol, such an append-only container suffers from revisit attack. If an agent visits collaborating malicious host n and m ($n < m$), host m can effectively remove the agent data from n to m without being detected. Another way to attack is to place a false set of data between host n and host m such that the data favor the malicious party. As long as the signatures for the fake data are valid, there is no way the butler can find out if the agent really visited those hosts. From these attacks on existing research, the importance of protecting agent itinerary is obvious. In SADIS, agent's itinerary is implicitly updated in the agent butler during key seed negotiation. This prevents any party from modifying the itinerary recorded on the butler and guard against all itinerary related attacks.

There is one recent research on agent data integrity protection called One-Time Key Generation System (OKGS) developed in Kwang-Ju Institute of Science and Technology, South Korea [13]. OKGS proposed an innovative approach of using a one-time data encryption key to encrypt the data provided by the host, and chain the encryption key to the hash values carried by the agent. When agents roam from host to host, each of them carry a hash value C_{i-1}. When the agent reaches host *i*, the host will generate two random values R_1 and R_2. It will perform an XOR operation on C_{i-1} and R_1, and hash the output to product data encryption key S_i. This data encryption key will be used to encrypt the data provided by the current host *i*. Subsequently, it will perform another XOR operation on the data encryption key S_i and R_2. The output of the XOR operation is hashed to become the next hash value C_i. The two random R_1 and R_2 will be encrypted together with the digital signature on the data using the agent butler's public key. When the agent returns to the butler, the butler can repeat the key derivation process to derive the data encryption key.

OKGS does protect the agent data against a number of attack scenarios under revisit attack, such as data insertion attack and data modification attack to certain extent. However, it does not protect the agent against deletion attack as two collaborating malicious hosts can easily remove roaming records in-between them.

Furthermore, the use of XOR operation and two different random values are identified as a main weakness of the algorithm. Firstly, XOR operation is subjected to easy manipulation if one party has control over one of the inputs and has knowledge about the others. In this case, the host can adjust the random value in such a way that the output of the XOR operation can be exactly what it wants. As a result, the host will be able to dictate the data encryption key to be used. Similarly, the host also has full control over the next hash value. Secondly, the use of two different random values does not introduce more randomness to the algorithm. On the contrary, given the vulnerability of the XOR operation earlier, using two random values gives the host more room for manipulation over the data encryption key and hash value. For example, a host can change its encryption key after the agent has left (e.g., when the agent reaches one of host's collaborating partners). By producing a different R_1, the host can change the encryption key to a different value. And by producing a suitable R_2, the hash value chaining effect can be maintained.

However, it should be pointed out that the signature algorithm does prevent the host from manipulating the encrypted data even though it can manipulate the encryption key. As a result, in case of this attack, the butler will probably find that the data provided by the host is corrupted but all chained signatures are valid. The data integrity is thus corrupted without being detected.

Instead of using a random value to generate data encryption key, SADIS makes use of a negotiated key seed to generate data encryption key. The advantage of this approach is that no random value needs to be encrypted and stored with the agent. This effectively reduces one PKI operation (encrypt the random value with the butler's public key) and optimizes the agent data size (does not need to carry encrypted random value any more). In addition, with the new design in SADIS, the weaknesses related to the XOR operation and two random values are not inherited.

Inspired by OKGS's innovative one-time encryption key concept, SADIS will extend this property to the communication between agent and butler as well. Not only is the data encryption key one-time, but the communication session key is as well. Using efficient hash calculations, the dynamic communication session key can be derived separately by the agent butler and the agent with minimum overhead. Despite the fact that all keys are derived from the same session-based key seed, SADIS also ensures that there is little correlation between these keys. As a result, even if some of the keys are compromised, the key seed will still remain secret.

There has also been some previous research under the SAFER initiative [1] that addresses agent integrity. Code-on-demand agent integrity protection [6] focuses on protecting the agent code integrity, including both static code and dynamic code (code-on-demand). It works together with SADIS to protect different component of an agent. Another earlier research on Agent Monitoring Protocol (AMP) [5] effectively addresses the issue of agent data integrity. The research itself is inspired by some existing limitation of data integrity protection mechanisms under revisit attack. Unlike some existing proposals in the literature, AMP does not require some trusted third party for its operation, thus reducing the infrastructure overhead to support agent roaming. Instead, the agent communicates with the butler actively to update the butler of its data, hash value, and itinerary whenever the agent roams from host to host. It is a protocol that is specifically designed to protect agents against revisit attack from malicious hosts. Unfortunately, the extensive use of PKI operations resulted in significant computation overhead. For each message sent between the butler and the agent, there are two PKI operations involved. Furthermore, it deposits its data and hash into the butler whenever it roams, thus requiring very active and heavy communication between the butler and the agent. In addition to protecting the agent from malicious attack, SADIS is designed with the objective of efficiency. For example, the use of PKI to protect agent to butler communication is replaced by symmetric key encryption and a new key seed negotiation protocol is introduced. Comparing with existing literature, SADIS is among the few that has focus not only on security, but efficiency as well, resulting in a highly practical solution.

2.2 Related Work on Agent Transport Protocol (SAT)

There has been a lot of research on the area of intelligent agents in the literature. Some only proposed certain features of intelligent agents, while others attempt to define a complete agent architecture. Unfortunately, there is no standardization in the various proposals, resulting in vastly different agent systems. Efforts are made to standardize some aspect of agent systems so that different systems can inter-operate with each other. In the area of knowledge representation and exchange, one of the most widely accepted standards is KQML [23] [32] (Knowledge Query and Manipulation Language) developed as part of the Knowledge Sharing Effort. KQML is designed as a high level language for runtime exchange of information between heterogeneous systems.

Unfortunately, KQML is designed with little security considerations because no security mechanism is built to address even the common security concerns, not to mention specific security concerns introduced by certain agent features. Agent systems using KQML will have to implement security mechanisms on top of KQML to protect itself. In an attempt to equip KQML with 'built-in' security mechanisms, Secret Agent [24] is proposed by Thirunavukkarasu.

Secret Agent defines a security layer on top of KQML. Applications will have to implement special message format in order to make use of Secret Agent. However, the solution does not gain much popularity. Secret Agent has a number of shortcomings and is handicapped by the design of KQML. Firstly, one requirement of Secret Agent is that every agent implementing the security algorithm must possess a key (master key). This master key is either a symmetric key or based on PKI. If the key is based on a symmetric key algorithm, due to the nature of symmetric key algorithm (encryption key and decryption key are the same), an agent will have to maintain a separate key with each agent it wishes to communicate. The prerequisite for an agent to communicate with another is that both of them have the knowledge of a common master key, which is exclusive to both of them. This requirement inevitably introduces the problem of key management. The maintenance as well as protection of the master key database may pose additional security threats to agent systems. For example, if the key database of Alice is compromised, all agents corresponding with Alice will be compromised. The point of failure for an agent is any one of the agents it corresponds with. If either one of them is compromised, the agent is compromised.

Furthermore, if the agent intends to talk to an agent with whom it has no common pre-established master key, a central authentication server is required to generate such a key. The use of a central authentication server introduces other issues into the architecture. Among them are potential attacks on the authentication server, key transport/exchange algorithm, key database management at the central authentication server etc.

If the master key is based on PKI, the agent identity must be tightly tied to the key pair. This was insufficiently addressed in the Secret Agent design, subjecting the algorithm to man-in-the-middle attack. For example, when agent Alice and Bob starts a handshake, if a third agent Eva can intercept all messages between Alice and Bob, agent Eva can pretend to be agent Alice while talking to agent Bob, and vice versa. If key and ID is not tightly integrated (like that in digital certificates), there is almost no way agent Alice or Bob can detect this attack. In the SAFER transport protocol, agent identity and key pair is tightly integrated using digital certification.

A number of limitations of Secret Agent are inherited from KQML [24]:

Limitation 1: Message delivery must be reliable and in order. This is because KQML assumes the message delivery is robust. In the SAFER transport protocol, there is no such assumption so the system can operate across heterogeneous systems.

Limitation 2: It does not support non-repudiation on receipt of messages due to the asynchronous nature of KQML. A receiving agent can safely deny the receipt of messages. However, there is no way an agent can deny similar events in the SAFER transport protocol.

Limitation 3: There is no support for exchanging credentials. Agents are unable to present their credentials to each other for verification. On the contrary, under the SAFER transport protocol, digital certificate exchange is frequently used in agent handshaking process.

Limitation 4: There is no support for replay detection if message ID method is not used in Secret Agents. The design of SAFER messages prevents such attack from taking place.

Another prominent transportable agent system is Agent TCL developed in Dartmouth College [18][33]. Agent TCL addresses most areas of agent transport by providing a complete suite of solutions. It is probably one of the most complete agent systems under research. Its security mechanism aims at protecting resources and the agent itself. Since some existing agent systems are already very strong in this area, Agent TCL 'seeks to confirm their sufficiency and either copy or redesign as appropriate' [28]. In terms of agent protection, the author acknowledges that 'it is clear that it is impossible to protect an agent from the machine on which the agent is executing… it is equally clear that it is impossible to protect an agent from a resource that willfully provides false information' [28]. As a result, the author 'seeks to implement a verification mechanism so that each machine can check whether an agent was modified unexpectedly after it left the home machine' [28]. In other words, it addresses agent integrity and provides certain level of traceability to the agents. The other areas of security, like non-repudiation, verification, identification, are not carefully addressed.

Besides Agent TCL, TACOMA is another agent system under active research. It is jointly developed by the University of Tromsø, Cornell University and the University of California, San Diego. The security focus of TACOMA is on fault tolerance. Agents are protected against faulty hardware under two protocols. Similar to Agent TCL, these protocols are based on assumptions of common secret or prior knowledge in source and destination. The use of hardware solution and requirement for pre-established shared secrets makes it difficult to achieve in the Internet environment. To protect a host from malicious agents, three approaches are proposed, one based on hardware, one using sandbox restriction and the other using proof-carrying code. While the hardware-based approach is almost impossible to enforce in a multi-vendor environment, the other two approaches are purely software based. They will be refined and enhanced in the SAFER transport protocol.

Compared to the various agent systems discussed above, SAFER is designed to address the special needs of e-commerce. The other mobile agent systems are either too general or too specific to a particular application. By designing SAFER with e-commerce application concerns in mind, the architecture will be suitable for e-commerce application. The most important concern is probably security as discussed in previous sections. Due to the nature of e-commerce, security becomes a prerequisite for any successful e-commerce application. Other concerns are mobility, efficiency and interoperability. In addition, the design allows certain flexibility to cater to different needs of different applications.

There is, however, an industrial strength generic transport protocol available in the literature. SSL, or Secure Socket Layer, is the generic transport protocol widely accepted and used in the Internet environment [36]. SAFER framework does not make use of SSL as its underlying transport mechanism for a number of reasons. Firstly, while there is no doubt about the security strength of SSL, it does not address certain concerns specific to SAFER (since it is a generic protocol that focus on transport security only). The implicit authentication of destination host by agent owner is one such example. If SSL is adopted, the authentication above must be carried out explicitly as it is not part of SSL, thus requiring a separate communication between the agent butler and destination host. However, with a tightly integrated transport protocol, the agent butler will be able to perform implicit authentication to the destination host through the specially designed transport protocol, improving its efficiency. Furthermore, a customized transport protocol allows other SAFER modules to have tighter integration with the transport mechanism to achieve higher efficiency. For example, although SADIS is designed to protect agent data integrity at destination host, it has dependencies on agent transport for its key seed negotiation process. Another limitation of SSL is that it treats all messages equally, regardless of the sensitivity of the messages. However, not all messages in the agent transport process are sensitive and require sophisticated encryption. With a customized solution, messages can be scrambled only if necessary to achieve optimal efficiency. From the various concerns discussed earlier, although SSL is a widely accepted industry standard for communication, SAFER does not simply use it as its transport mechanism.

Chapter 3

SADIS

3.1 Overview

SADIS is designed based on the SAFER framework. The proposal itself is based on a number of assumptions that was implemented under SAFER.

Firstly, entities in SAFER, including agents, butlers and hosts, should have globally unique identification number (IDs). This ID will be used to uniquely identify each entity.

Secondly, each agent butler and host should have a digital certificate that is issued by a trusted CA under SAFER. These entities with digital certificate will be able to use the private key of its certificate to perform digital signatures and, if necessary, encryption.

Thirdly, while the host may be malicious, the execution environment of mobile agents should be secure and the execution integrity of the agent can be maintained. This assumption is made because protecting the agent's execution environment is a completely separate area of research that is independent of this thesis. There are some discussions in this area in the literature. Without secure execution environment and execution integrity, none of the agent data protection scheme will be effective.

Another assumption is that entities involved are respecting and cooperating with the SADIS protocol. For example, where digital signature is required, the signer should be willing to perform the signature under the protocol. Otherwise, the refusal will be immediately detected by the requestor and the entity refusing to sign will be excluded from future interactions with the requestor. As a result, there is no incentive for any entity to refuse full cooperation under the SADIS protocol.

Given the fact that the agent may be executing in a malicious environment, and that even if the execution integrity is maintained, the privacy of the execution may not be guaranteed, SADIS does not require the agent to carry any private key with it. As a result, there is no need for the agent to carry any digital certificates. Agent authentication can be achieved by verifying the digital signature on the agent code (agent code integrity protection).

Lastly, SADIS does not require the agent to have a pre-determined itinerary. The agent is able to decide which is the next destination host independently.

Under SADIS, data integrity and agent-to-butler communication are protected by a session-based key seed. This key seed will be negotiated between the agent and butler every time the agent roams to a new host and will remain valid throughout the agent's visit to the host. A one-time data encryption key will be derived from the key seed to encrypt data provided by the current host. The communication between the agent and the butler will be protected by communication session keys. Communication session key is also derived from the key seed using a different formula. Different from most conventional session keys, SADIS makes use of an evolving session key instead of a static session key. The formula for session key derivation contains a variable component that ensures session keys generated for each message exchange will be different from each other. At the same time, the formula is also designed to ensure that there is no correlation between the subsequent session keys through the use of hash functions. In this way, any attempt to attack the key through cipher-text attack will be extremely difficult since each message will be encrypted using a different key. The key seed negotiation protocol and various key generation algorithms will be discussed in detail in the next section.

The proposed key seed negotiation protocol lays the necessary foundation for the data integrity protection protocol. At the end of agent roaming, the host will provide a set of data to be carried by the agent. The host will also perform a digital signature on the current data as well as the signature from the previous host using its private key. The signature can be subsequently verified whenever the agent reaches a new destination or returns to the agent butler. The details of the data signature generation and integrity verification process will be discussed in details in data integrity protection protocol section.

3.2 Key Seed Negotiation Protocol

The proposed key seed negotiation protocol defines the process for key seed negotiation as well as session key and data encryption key derivation.

When an agent first leaves the butler, the butler will generate a random initial key seed, encrypt it with the destination host's public key and deposit into the agent before sending the agent to the destination host. It should be noted that the agent transmission is protected by the agent transport protocol [3]. Otherwise, a malicious host (man-in-the-middle) can perform an attack by replacing the encrypted key seed with a new key seed and encrypt it with the destination's public key. In this case, the agent and the destination host will not know the key seed has been manipulated. When the agent starts to communicate with the butler using the wrong key seed, the malicious host can intercept all the messages and re-encrypt them with the correct key derived from the correct key seed and forward them to the agent butler. In this way, a malicious host can compromise the whole protocol.

The key seed carried by the agent is session-based, it is valid until the agent leaves the current host. When the agent decides to leave the current host, it must determine the destination host, and start the key seed negotiation process with the agent butler.

The key seed negotiation process is based on the Diffie-Hellman (DH) key exchange protocol [15] with a variation. The agent will first generate a private DH parameter a and its corresponding public parameter x. The value x, together with the ID of the destination host, will be encrypted using a communication session key and sent to the agent butler.

The agent butler will decrypt the message using the same communication session key (derivation of communication session key will be discussed later in the section). It too, will generate its own DH private parameter *b* and its corresponding public parameter *y*. With the private parameter *b* and the public parameter *x* from the agent, the butler can derive the new key seed and use it for communications with the agent in the new host. Instead of sending the public parameter *y* to the agent as in normal DH key exchange, the agent butler will encrypt the value *y*, host ID, agent ID and current timestamp with the destination host’s public key to get message *M*. Message *M* will be sent to the agent after encrypting with the communication session key.

$$M = E(y + ID_{host} + ID_{agent} + timestamp, H_{pubKey}) \qquad (1)$$

At the same time, the agent butler updates the agent’s itinerary and stores the information locally. Since the agent itinerary is stored locally in SADIS, it effectively protects the agent’s actual itinerary against any hacking attempts related to itinerary. The protection of agent itinerary in turn, protects the agent against certain data integrity attack, namely, data deletion attack.

When the agent receives the double-encrypted DH public parameter *y*, it can decrypt with the communication session key. Since the decrypted result *M* is parameter *y* and some other information encrypted with the destination host's public key, the current host will not be able to find out the value of *y* and thus find out the new key seed to be used when the agent reaches the destination host. It should be noted that this does not prevent the host from replacing *M* with its own version *M’* with the same host ID, agent ID, timestamp but different *y*. The inclusion of host ID, agent ID inside *M* can render such attack useless against SADIS. A detailed discussion on this attack can be found in the security analysis section.

Subsequently, the agent will store *M* into its data segment and requests the current host to send itself to the destination host using the agent transport protocol [3].

On arriving at the destination host, the agent will be activated. Before it resumes normal operation, the agent will request the new host to decrypt message *M*. If the host is the correct destination host, it will be able to use the right private key to decrypt message *M*, and thus obtain the DH public parameter *y*. As a result, the decryption of message *M* not only completes the key seed negotiation process, but also serves as a means to authenticate the destination host. Once the message *M* is decrypted, the host will verify that the agent ID in the decrypted message matches the incoming agent, and the host ID in the decrypted message matches that of the current host. In this way, the host can ensure that it is decrypting for a legitimate agent instead of some bogus agent (this is to prevent an attack scenario depicted in the security analysis section). If the IDs in the decrypted messages match, the decrypted value of *y* is returned to the agent.

With the plain value of *y*, the agent can derive the key seed by using its previously generated private parameter *a*. With the new key seed derived, the key seed negotiation process is completed. The agent can resume normal operation in the new host.

Whenever the agent or the butler needs to communicate with each other, the sender will first derive a communication session key using the key seed and use this communication session key to encrypt the message. The receiver can make use of the same formula to derive the communication session key from the same key seed to decrypt the message.

The communication session key K_{CSK} is derived using the formula below:

$$K_{CSK} = Hash(key\_seed + ID_{host} + seqNo) \qquad (2)$$

The sequence number is a running number that starts with 1 for each agent roaming session. Whenever the agent reaches a new host, the sequence number will be reset to 1. In this way, each message communicated will be encrypted using a different key. Given the varying communication session key, if one of the messages is somehow lost without being detected, the butler and agent will not be able to communicate afterwards. As a result, SADIS makes use of TCP/IP as a communication mechanism so that any loss of messages can be immediately detected by the sender. In the case of an unsuccessful message, the sender will send ‘ping’ messages to the recipient in unencrypted format until the recipient or the communication channel recovers. Once the communication is re-established, the sender will resend the previous message (encrypted using the same communication session key). In this way, the agent and the butler can synchronize on communication session key calculations.

When the host provides information to the agent, the agent will encrypt the information with a data encryption key K_{DEK}. The data encryption key is derived as follows:

$$K_{DEK} = Hash(key\_seed + ID_{host}) \qquad (3)$$

The details on encryption will be discussed in the next section.

3.3 Data Integrity Protection Protocol

The key seed negotiation protocol lays the necessary foundation for integrity protection by establishing a session-based key seed between the agent and its butler. Agent data integrity is protected through the use of this key seed and the digital certificates of the hosts. This section will illustrate the data integrity protection protocol in details.

Our data Integrity Protection protocol is comprised of two parts: chained signature generation and data integrity verification. Chained signature generation is performed before the agent leaves the current host. The agent gathers data provided by the current host d_i and construct D_i as follows:

$$D_i = E(d_i + ID_{host} + ID_{agent} + timestamp, k_{DEK}) \qquad (4)$$

Or,

$$D_i = d_i + ID_{host} + ID_{agent} + timestamp \qquad (5)$$

The inclusion of host ID, agent ID and timestamp is to protect the data from possible replay attack, especially when the information is not encrypted with the data encryption key. For example, if the agent ID is not included in the message, a malicious host can potentially replace the data provided for one agent with that provided for a bogus agent. Similarly, if timestamp is not included into the message, earlier data provided to the same agent can be used at a later time to replace current data provided to the agent from the same host. The inclusion of the IDs of the parties involved and a timestamp essentially creates an unambiguous memorandum between the agent and the host.

Note that the construction of D_i gives the flexibility to encrypt the data or keep it in plain. As far as the agent integrity protection protocol is concerned, it does not matter whether the data is encrypted (since the data integrity is protected using chained digital signature). The agent butler or the agent itself can decide if the data should be encrypted. As a general rule of thumb, it is recommended that the agent should encrypt data that is not required for the remaining of the roaming session for maximum security.

After constructing D_i, the agent will request the host to perform a signature on the following:

$$c_i = Sig(D_i + c_{i-1} + ID_{host} + ID_{agent} + timestamp, k_{priv}) \qquad (6)$$

where c_0 is the digital signature on the agent code by its butler.

There is some advantage with the use of a chained digital signature compared to the conventional signature approach. In the scenario when a malicious host attempts to modify the data from an innocent host *i* and somehow manages to produce a valid digital signature c_i, the data integrity would have been broken if the digital signature is independent and not chained to each other. The independent digital signature also opens the window for host *i* modify data provided to the agent at a later time (one such scenario is the agent visits one of the host's collaborating partners later). Regardless of the message format used, so long as the messages are independent of each other, host *i* will have no problem reproducing a valid signature to the modified message. In this way, data integrity can be compromised. With chained digital signature, even if the malicious host (or host *i* itself) produces a valid digital signature after modifying the data, the new signature c_i' is unlikely to be the same as c_i. If the new signature is different from the original signature, as the previous signature is provided as input to the next signature, the subsequent signature verification will fail, thus detecting compromise to data integrity. The inclusion of host ID, agent ID, and timestamp prevents anyone from performing a replay attack.

When the agent reaches a new destination, the host must perform an integrity check on the incoming agent. In the design of SADIS, even if the new destination host does not perform an immediate integrity check on the incoming agent, any compromise to the data integrity can still be detected when the agent returns to the butler. The drawback, however, is that the identity of the malicious host may not be established. One design focus of SADIS is not only to detect data integrity compromise, but more importantly, to identify malicious hosts. To achieve malicious host identification, it is an obligation for all hosts to verify the incoming agent's data integrity before activating the agent for execution. In the event of data integrity verification failure, the previous host will be identified as the malicious host.

Data integrity verification includes the verification of all the previous signatures. The verification of signature c_0 ensures agent code integrity, the verification of c_i ensures data provided by host h_i is intact. If any signature failed the verification, the agent is considered compromised.

While the process to verify all data integrity may seem to incur too much overhead and somewhat redundant (e.g., why need to verify the integrity of d_1 in h_3 while host h_2 already verifies that), it is necessary to ensure the robustness of the protocol and to support the function of malicious host identification. For example, if only the signatures of the n consecutive previous hosts are verified, in the scenario when the previous n consecutive hosts happen to be all malicious and collaborating with one another, these malicious hosts can somehow produce the illusion to the next innocent host that data integrity has been maintained by creating seemingly correct signatures. As the next innocent host only verifies the previous n signatures that happen to be the creation of malicious hosts, it will get the impression that data integrity has not been compromised. Although the agent butler can eventually detect such data integrity compromise (since agent butler has to verify all signatures), there is no way to establish the identity of malicious host(s).

Chapter 4

SADIS Analysis and Prototype

4.1 Security Analysis

To analyze the effectiveness and reliability of SADIS, a detailed security analysis is performed subjecting SADIS to a variety of attacks. Based on the targets and types of attacks, the various attacks to SADIS can be classified into data attack, key attack, signature attack, itinerary attack, and composite attack. Composite attack refers to attacks that are combinations of one or more of the above-mentioned attacks. The security analysis will be organized according to the above classifications.

4.1.1 Data Attack

Data attack refers to any attempt that aims to compromise the data carried by an agent. Compromise can be in the form of data modification, deletion, or insertion.

Let us consider the scenario of data modification where a malicious host wants to modify agent data or one of the hosts in the agent itinerary attempts to modify its own data after the agent has left. Assume the data targeted is D_i provided by host *i*, since the agent itinerary is protected by the butler and cannot be changed, only host *i* can produce a valid signature if the data were to be modified. However, even if the malicious party (or even host *i* itself) can produce a valid signature c_i' corresponding to D_i', since c_i is chained to the signature of the next host c_{i+1}, signature verification for host *(i+1)* will fail. If the malicious host wants to ensure the signature verification for the next host is also successful, it has to forge the signature of the next host as well. Following similar argument, in order to perform a successful data modification attack, the malicious host must be able to forge the signatures for all hosts in the itinerary since host *i*. As the only way to achieve this is to obtain the private keys of all the following hosts, data modification attack is extremely difficult under SADIS.

Another way to compromise data integrity is by inserting additional data into the agent. This includes inserting into data provided by hosts in the agent itinerary as well as inserting new hosts into the existing itinerary and fabricating data from the new host. The former scenario is the same as data modification attack. In the second scenario, the malicious host essentially needs to modify the itinerary of the agent. This will be covered in the discussion on itinerary attack later in the section.

Other than data modification and data insertion, data deletion is another form of data integrity attack. As illustrated in the discussion in related work, quite a number of the existing data integrity protocols suffer from this attack. After analyzing the root cause of the vulnerabilities, it is realized that it's extremely important to protect the agent's itinerary. Otherwise, in the case of a revisit attack, the subsequent host can easily 'restore' the agent to the state of its previous visit to one of the host's collaborator in the agent's itinerary. However, if the agent's itinerary is closely guarded by the butler, any data deletion will result in modification to the agent's itinerary and thus be detected.

4.1.2 Key Attack

Besides direct attack on data integrity, a malicious host may attempt to attack the various keys in order to compromise data integrity. There are three different types of keys in SADIS. They are session-based key seed, communication session key, and data encryption key.

In SADIS, a key seed is negotiated between the agent and the butler during agent roaming process. Once the key seed is negotiated, it will be kept by the agent and the butler separately. It will not be used directly as encryption key at all. Attacks to the key seed can only target the key seed negotiation protocol. As all communication in key seed negotiation is protected by the communication session key, we can safely rule out the possibility of any third party malicious attempts to break the protocol. We can focus on the scenario where the current host attempts to break the key exchange to obtain the key seed to be used in the subsequent host. Given the simplicity of DH key exchange, the parameters available for manipulation is the DH private parameter a in plain text and the encrypted DH public parameter from butler y encrypted using the destination host's public key.

Firstly, without any manipulation, the current host will not be able to complete DH key exchange to find out the new key seed. This is because the DH public parameter from butler *y* is encrypted using the destination host's public key. Without the private key from the destination host, no one can obtain *y* to complete the key exchange. Furthermore, as the encrypted message contains the agent ID and destination host ID, the current host won't be able to send a bogus agent carrying this encrypted *y* to the destination host for decryption.

If the current host attempts to manipulate any one or both of these parameters, it is able to manipulate the key seed derived when the agent reaches the destination host. (This is because any change to *a* or *y* will change the result of key exchange, and anyone can forge the encrypted *y* since the encryption key is a public key). However, the change in key seed will be immediately detected when the agent communicates with the butler or vice versa. This attack can only change the key seed in the agent but won't be able to compromise the key seed in the butler. In order to perform a successful attack, the current host must also be able to obtain the key seed in the butler so that it can act as a middle-man subsequently to intercept and replace message communicated between the butler and the agent. Unfortunately, as illustrated earlier, there is no way the current host can find out the value of DH public parameter from butler *y*. Thus, the key seed will not be compromised.

Besides key seed, SADIS makes use of communication session key and data encryption key in the protocol. These two keys are directly derived from the session-based key seed using a hash function. In the case of communication session key, a sequence number is used in the key derivation to ensure each message communicated is encrypted with a different and unrelated communication session key. As far as any third-party host is concerned, attack to communication session key or data encryption key is equivalent to attacking the encryption key given only the cipher text. Even in the extreme case when such a key is compromised, the loss is limited to the message it encrypts. The other keys will remain in secret due to the nature of one-way hash functions.

4.1.3 Signature Attack

Despite being categorized separately, signature attack is meaningless if carried out alone. Usually a malicious host would need to forge digital signature when it attempts to compromise data integrity. If data integrity is not compromised, there is no need to attack the chained signature at all. Signature related attacks due to data integrity compromise have been discussed earlier in the section.

4.1.4 Itinerary Attack

At the first glance, agent itinerary may not seem highly sensitive. However, as examination of related work shows, if agent itinerary is not carefully protected, it may lead to compromise to data integrity, especially in the case of data deletion as illustrated earlier in the section. Given the importance of agent itinerary protection, SADIS employs a relatively conservative approach to protecting agent itinerary by storing the itinerary information in the butler as the agent roams. As the agent updates the butler of its next destination host as part of the key seed negotiation protocol, there is no additional overhead related to the itinerary protection mechanism. With the agent itinerary updated and stored with the agent butler, there is no way a malicious host can perform any attack on the itinerary (except, of course, if it breaks into the agent butler).

4.1.5 Composite Attack

As the analysis above shows, agent data integrity attack may not always target only in one area. At times, in order to perform a successful attack, more than one area are targeted simultaneously. These composite attacks have been discussed in the earlier section along with analysis on different attack targets.

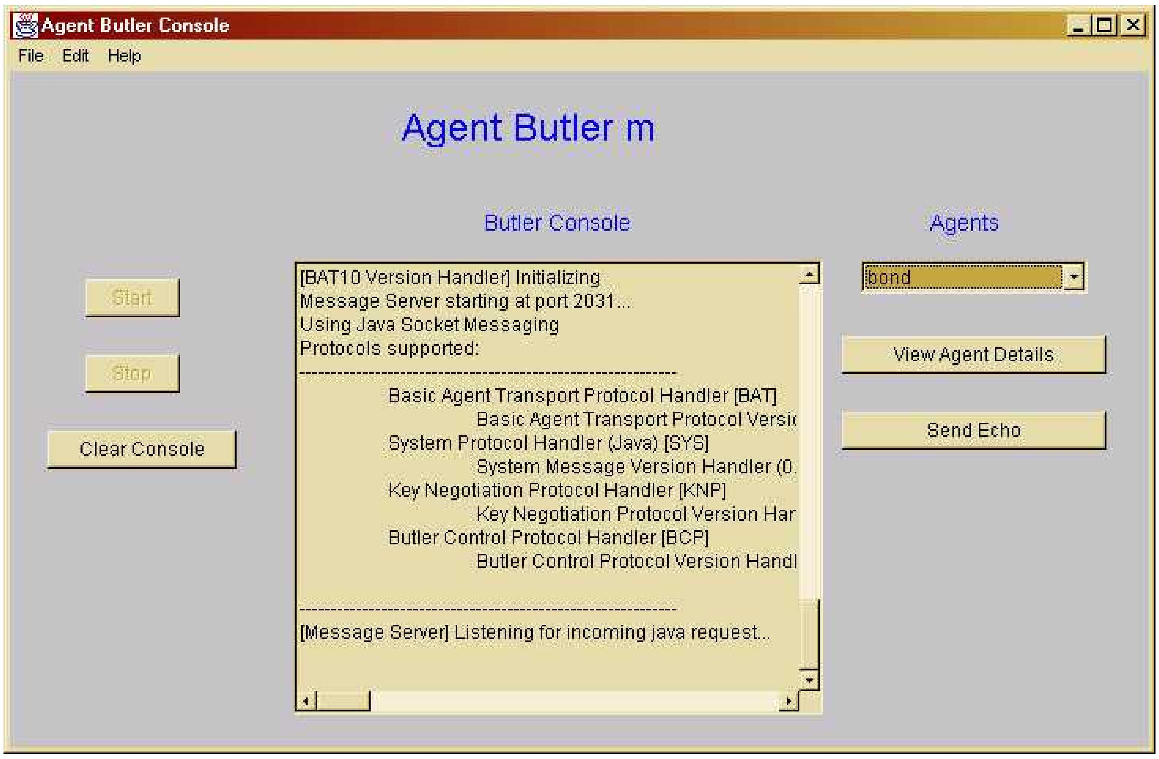
In addition to attacks with specific targets, there are certain general hacking techniques such as man-in-the-middle attack, replay attack. The design of SADIS employs a mechanism to protect the protocol against these hacking techniques. Through the use of communication session key, man-in-the-middle attack can be avoided (This is because man-in-the-middle attack will not be effective if the attacker can’t decrypt the message at all). On the other hand, the use of sequence number in communication session key generation effectively protects the protocol from replay attack by a third party host. In addition, the inclusion of host ID, agent ID, and timestamp during the key seed negotiation process prevents the current host from performing a replay attack with the next destination host (attempting to obtain the next key seed).

Lastly, the design of SADIS does not have dependency on any specific encryption/hashing algorithm. In an unlikely scenario when one algorithm is broken, SADIS can always switch to a stronger algorithm.

4.2 Implementation

In order to verify the design of SADIS and assess its applicability, a prototype of SADIS is developed. The prototyping language is chosen to be Java. One of the main reasons for choosing Java is its platform independent feature. Internet is a complex environment that comprises of various platforms. With Java as the prototyping language, the effort required to port the prototype from one platform to another can be avoided. Furthermore, being one of the leading programming platforms in the marketplace, Java has a wide range of libraries to choose from. Modules such as cryptographic library, messaging utility, etc. are already available to be used as components in the prototype. The reuse of existing modules significantly shortened the prototyping effort, allowing the team to put its main focus on the research.

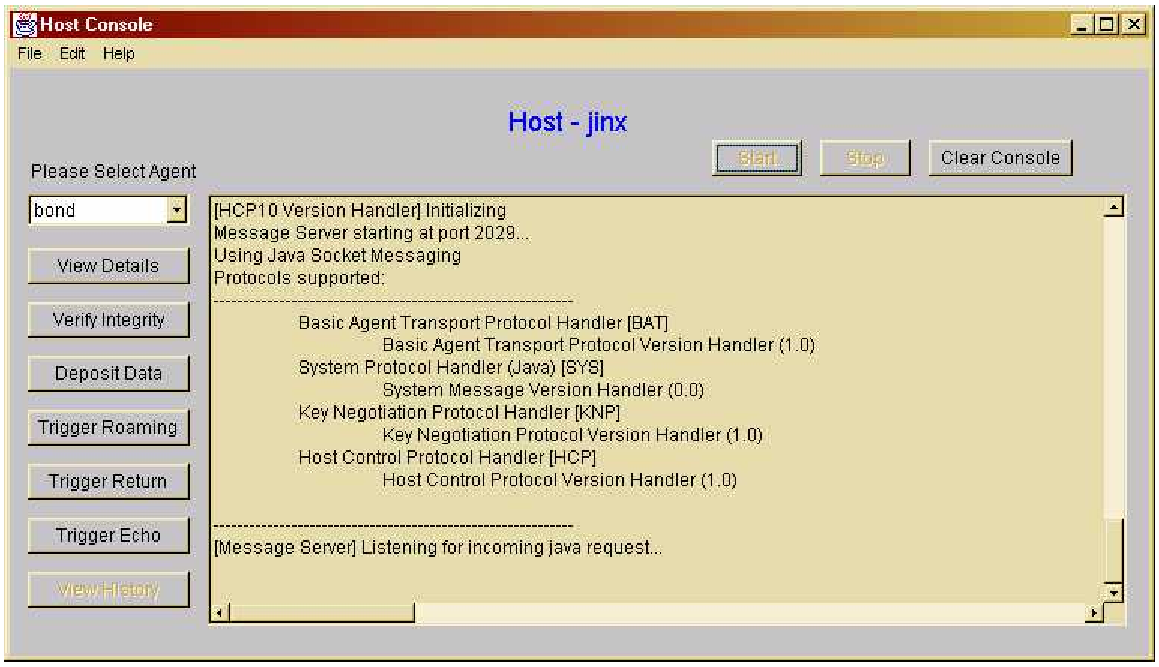
The prototype consists of four different entities: the agent butler m, agent *bond*, and two hosts *jinx* and *natalya*. The agent butler *m* (as shown in Figure 3) coordinates the agent's roaming (console of agent *bond* is shown in Figure 6), participates in key seed negotiation, tracks the agent's whereabouts and receives the agent during its return. Host *jinx* (as shown in Figure 4) plays the role of source host. It is the host where the agent is originally located. After agent *bond* (agent *bond*'s console is shown in Figure 6) completes its processing in *jinx*, it will get *jinx* to sign the data it collected from it. Once the signature is obtained, it will trigger the key seed negotiation process with butler *m* and roam to the destination host *natalya*. Upon arrival of agent *bond*, host *natalya* (as shown in Figure 5) will perform data integrity verification on the agent *bond* before assisting it to complete the key seed negotiation process. Once the key seed negotiation is completed, agent *bond* can resume its operation. To further illustrate the use of communication session key, the agent butler *m* and the agent bond can send messages to each other at any time. The communication session key will be synchronized between the two, ensuring each message is encrypted using a different key. At the end of the agent roaming, agent bond will return to butler *m*. In addition to performing data integrity check on the agent bond, *m* can also decrypt the data carried by bond using the various key seeds.



```
[BAT10 Version Handler] Initializing
Message Server starting at port 2031...
Using Java Socket Messaging
Protocols supported:
------------------------------------------------------------
            Basic Agent Transport Protocol Handler [BAT]
                        Basic Agent Transport Protocol Versi
            System Protocol Handler (Java) [SYS]
                        System Message Version Handler (0.
            Key Negotiation Protocol Handler [KNP]
                        Key Negotiation Protocol Version Har
            Butler Control Protocol Handler [BCP]
                        Butler Control Protocol Version Handl

------------------------------------------------------------
[Message Server] Listening for incoming java request...
```

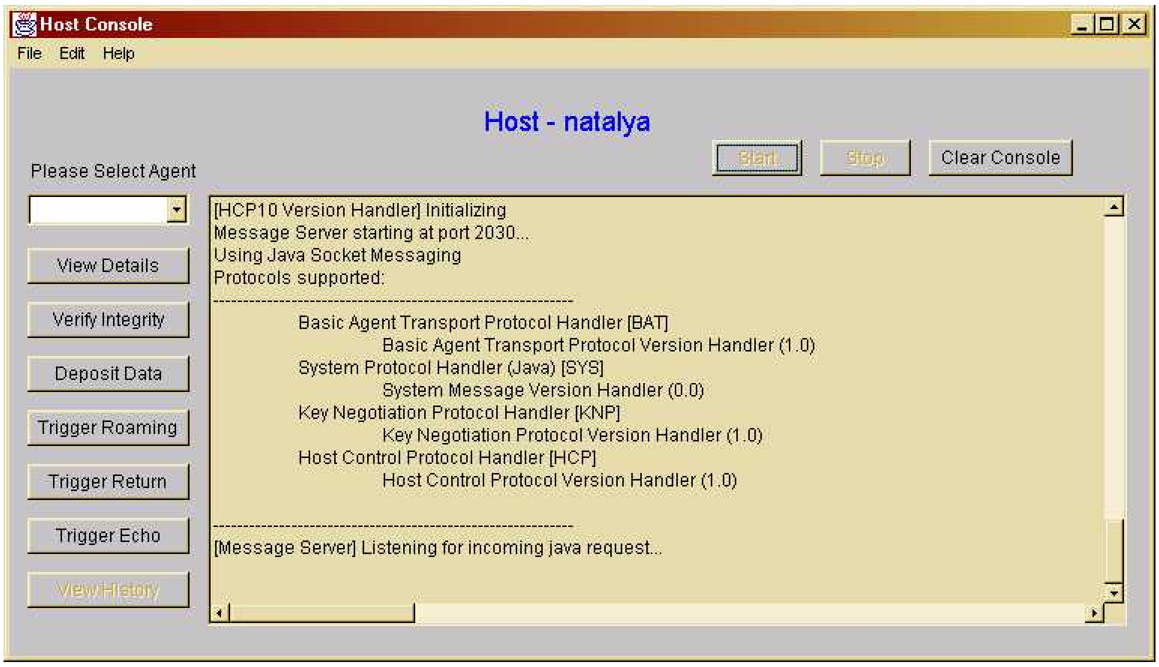
Figure 3 Agent butler console



```
[HCP10 Version Handler] Initializing
Message Server starting at port 2029...
Using Java Socket Messaging
Protocols supported:
------------------------------------------------------------
              Basic Agent Transport Protocol Handler [BAT]
                            Basic Agent Transport Protocol Version Handler (1.0)
              System Protocol Handler (Java) [SYS]
                            System Message Version Handler (0.0)
              Key Negotiation Protocol Handler [KNP]
                            Key Negotiation Protocol Version Handler (1.0)
              Host Control Protocol Handler [HCP]
                            Host Control Protocol Version Handler (1.0)

------------------------------------------------------------
[Message Server] Listening for incoming java request...
```

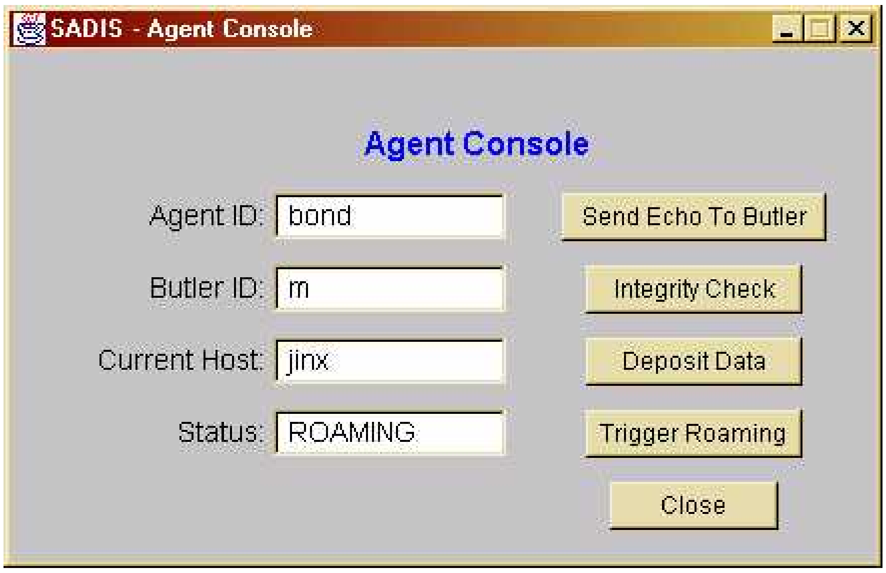
Figure 4 Host *jinx* console



```
[HCP10 Version Handler] Initializing
Message Server starting at port 2030...
Using Java Socket Messaging
Protocols supported:
------------------------------------------------------------
              Basic Agent Transport Protocol Handler [BAT]
                            Basic Agent Transport Protocol Version Handler (1.0)
              System Protocol Handler (Java) [SYS]
                            System Message Version Handler (0.0)
              Key Negotiation Protocol Handler [KNP]
                            Key Negotiation Protocol Version Handler (1.0)
              Host Control Protocol Handler [HCP]
                            Host Control Protocol Version Handler (1.0)

------------------------------------------------------------
[Message Server] Listening for incoming java request...
```

Figure 5 Host *natalya* console



Agent ID: bond
Butler ID: m
Current Host: jinx
Status: ROAMING

Figure 6 Agent *bond*'s console

Each entity in the prototype contains common data such as unique ID, description, parent ID (for entities without parents, this field can be null). Individual entities like agent, agent butler extends from the common entity and supplement with its specific data structures. The additional data structures for agent, agent butler, host, and other supporting entities are shown as follows:

```
Agent extends SAFEREntity {
        byte[] key_seed;                    // Session-based Key Seed
        int skey_counter;                   // Communication Session Key Counter
        String current_hostID;              // Current host ID
        IkeyExchange keyExchange;           // Key exchange protocol
        byte[] encrypted_y;                 // Encrypted DH parameter y
        AgentSuitcase agentSuitcase;        // Agent suitcase to store itinerary and
data
};

Butler extends SAFEREntity {
        IkeyExchange keyExchange;  // Key exchange protocol
        byte[] key_seed;           // Current key seed
        int skey_counter;          // Communication session key counter
        Hashtable local_agents;    // Agents currently residing in butler
        Hashtable roaming_agents;  // Agents currently roaming outside
        ItineraryTracker tracker;  // Agent itinerary tracker
};

Host extends SAFEREntity {
        Hashtable agents;     // Agents visiting current host
}

// AgentSuitcase stores the data collected by an agent from various hosts
AgentSuitcase {
        String agentID;       // Agent ID
        Vector datas;         // Data collected from various hosts
        Vector signatures;    // Signatures from various hosts
        Vector hosts;         // Hosts information
}

// ItineraryTracker assists the butler to keep track of all its agents' itineraries
```

```
ItineraryTracker {
        Hashtable currentHostIDs;    // IDs of the current hosts
        Hashtable currentHostAddrs;          // Host Name of current hosts
        Hashtable currentHostPorts;          // Port Number of current hosts
        Hashtable currentKeySeeds;           // Key seeds of current hosts
        Hashtable histories;                 // Histories for agents
}
```

The communication between entities is developed on top of a messaging server and uses TCP/IP as its underlying protocol. The messaging server is designed as a generic messaging platform that is capable of supporting multiple message protocols. In the earlier prototype of agent transport, the messaging server was used to support agent transport messaging protocol. In the current prototype, three more protocols are developed: key negotiation protocol, butler control protocol, and host control protocol. Key negotiation protocol (KNP) is in charge of handling the message exchanges during key seed negotiation. It is equipped by the agent butler as well as every host. Butler control protocol, (BCP), is used to control the butler during the prototype demonstration. Functions within butler control protocol include requesting the display of agent information, sending echo messages to agent, and requesting for agent’s return etc. Similarly, a host control protocol, (HCP), is developed to control hosts. Functions within host control protocol include sending agent to another host, receiving incoming agent, etc. The messages supported by each protocol are illustrated in Appendix C.

It can be seen in the various screenshots below that different protocols can be selectively loaded onto an entity based on requirement. For example, in Figure 3, it can be seen that agent butler is loaded with agent transport protocol (BAT), key negotiation protocol (KNP), and butler control protocol (BCP). The use of a generic messaging server facilitates the modularization of the prototype and integration of various protocols.

Just like any other security mechanism, there is certain overhead associated with SADIS. The overhead is incurred as additional time required for processing as well as additional data carried by the agent.

To assess the efficiency of SADIS, benchmarking is performed on the prototype. The benchmarking environment is composed of three PCs connected with each other in an intranet environment with 100MB LAN connections. One PC acts as the agent butler m, while the other two act as host *jinx* and *natalya* respectively. Agent bond travels between the three entities during the roaming and data collection simulation. Each PC is configured with PIII 800 MHz processor with 512MB RAM each. The result of SADIS benchmarking is broken down based on functionality and is shown in Table 1 and Table 2. From the tables, it can be seen that the bulk of the overhead is incurred during key seed negotiation where the key exchange protocol and the public key operation is performed. During key seed negotiation, one PKI operation is incurred in the agent butler when it encrypts the public parameter of the key exchange with the destination host’s public key, and another PKI operation when the destination host decrypts the incoming encrypted key exchange parameter. Given the computation intensive nature of PKI operation, it is expected that the overhead incurred during the key seed negotiation process will be relatively higher than the rest. Despite the relatively high overhead, this will not impact the overall performance of SADIS significantly because the frequency of agent roaming is low compared to the frequency of some other agent operations (such as agent to butler communication). As a result, the overhead incurred at this stage is 'one-time' in nature. Comparing with the statistics from OKGS, OKGS general incurs additional processing time of more than 500 milli seconds. Assuming there is 0 additional overhead caused by non-PKI operations, each PKI operation in OKGS incurs an overhead of 250 milli-seconds. In SADIS prototype, the overhead of one PKI operation is roughly 230 milli-seconds (take the average overhead of the key seed negotiation process). The two figures are very close to each other, suggesting a similar prototyping configuration. Coincidentally, this number is slightly more than twice the overhead in SADIS. Considering the fact that OKGS requires one more PKI operation in the message exchange, the statistics shows that SADIS' efficiency improvement when the use of PKI operations is minimized. The time savings achieved is the time taking for one PKI operation. In SADIS prototype, this is about 230 to 250 milli-seconds.

Table 1 SADIS Time Efficiency – Performance without SADIS

Operation	**1 (ms)**	**2 (ms)**	**3 (ms)**	**4 (ms)**	**5 (ms)**	**Avg (ms)**
Key Seed Negotiation (butler timing)	40	50	50	40	40	44.0
Key Seed Negotiation (destination host)	41	41	40	40	40	40.4
Agent Butler Communication (agent timing – send)	40	40	50	40	40	42.0
Agent Butler Communication (butler timing – send)	30	30	31	40	30	32.2
Agent Butler Communication (agent timing – receive)	10	10	10	10	10	10.0
Agent Butler Communication (butler timing – receive)	10	30	10	10	20	16.0

Table 2 SADIS Time Efficiency – Performance Comparison with SADIS

Operation	1 (ms)	2 (ms)	3 (ms)	4 (ms)	5 (ms)	Avg (ms)	Overhead (ms)
Key Seed Negotiation (butler timing)	250	260	250	220	260	248.0	204.0
Key Seed Negotiation (destination host)	290	281	260	280	290	280.2	239.8
Agent Butler Communication (agent timing – send)	60	60	70	50	60	60.0	18.0
Agent Butler Communication (butler timing – send)	41	50	40	40	40	42.2	10.0
Agent Butler Communication (agent timing – receive)	10	20	10	10	10	12.0	2.0
Agent Butler Communication (butler timing – receive)	30	30	30	20	20	26.0	10.0

Other than in the key seed negotiation, the time overhead incurred elsewhere in the protocol is negligible. As shown in the two tables, with the key seed negotiated, the time overhead incurred during message exchange will not exceed 20 milli-seconds. This is due to the use of Symmetric-Key Encryption during the more frequent message exchanges. The efficiency of the evolving communication session key can also be shown statistically as its contribution to the time overhead is negligible.

Other than overhead in terms of processing time, there is certain overhead to data size as well. Before the detailed analysis of data overhead, it is necessary to point out that SADIS is designed to produce almost fixed data overhead regardless of the data size. In another word, regardless of the size of actual data, the overhead associated with SADIS is almost fixed, and can be limited to a fixed number of bytes. As a result, SADIS tends to be more efficient when actual data size is higher. While some of the existing literature also achieves higher efficiency when data size increase (e.g., OKGS), the size of the overhead increases when the size of actual data as well. The larger the data size, the higher the overhead size. However, the data overhead is SADIS has a maximum size regardless of the data size and does not increase as the data size increases. This ability to limit the size of overhead data regardless of actual data size is a significant improvement in efficiency over existing work.

The various overheads of SADIS can be best illustrated in Table 3. The first data overhead is incurred during the padding for symmetric key encryption. As most popular symmetric key encryption algorithm works on fixed length data blocks, it is necessary to pad the plain data into multiples of the block size before performing the encryption. The symmetric algorithm used in the current prototype is triple-DES that operates on blocks of 8 bytes. As a result, the padding will produce a maximum of 8 bytes data overhead.

Another data overhead is in the generation of data *Di*. For security purposes, the IDs of the host and agent are added to the actual data together with the current timestamp. The prototype makes use of Java type ‘Long’ to model the IDs. And the timestamp is also a ‘Long’ in Java. Since each ‘Long’ occupies 8 bytes of storage space, the total overhead will be 24 bytes.

The last and most significant overhead is the digital signature created by the host. While the actual size of the digital signature depends on the signing algorithm used, the size of the digital signature is always a fixed length. In our prototype, RSA is used as the digital signature algorithm. Thus, the overhead of digital signature is a fixed length of 64 bytes.

Altogether, SADIS has a maximum data overhead of 96 bytes. Assuming the actual data size is 1800 bytes (this is smallest actual data size used in the benchmarking of OKGS), this yields a data overhead of 5.33%. This figure will improve linearly as the size of the actual data increases. The data overhead of 5.33% is compared with the benchmark of OKGS that averages to 36.2% (actual data size in OKGS is from 1836 to 2001).

Table 3 SADIS Data Overhead

	Original Data Size	Maximum Overhead	Overhead	OKGS Overhead
1	1800	96	5.33%	33.87%
2	2001	96	4.80%	37.73%
3	5000	96	1.92%	N/A
4	10000	96	0.96%	N/A
5	100000	96	0.10%	N/A

As the statistics shows, SADIS is optimized to improve both time efficiency instead of data efficiency compared with related work in the literature. The feasibility and practicality of SADIS are thus demonstrated through the prototype.

4.3 Attack Simulation

4.3.1 Overview

In order to further verify the security strength of SADIS protocol, various attack scenarios are simulated using the prototype developed in previous section. Five different attack scenarios covering all attacks described in the security analysis section are simulated. The simulations and their results are discussed in this section.

4.3.2 Data Modification Attack

a. Scenario Description

The data modification attack simulates the situation when a malicious host attempts to modify data provided by another host. In the simulation for this attack, the data provided by host *jinx* to agent *bond* is modified by an unknown malicious host during agent roaming. As data carried by bond is already encrypted with the communication session key corresponding to the roaming session with *jinx*, the malicious host can only modify part of the encrypted message in order to confuse the agent butler (data is decrypted when the agent returns to agent butler) or the subsequent hosts (data integrity check is performed at every host).

In this case, the compromise to agent data integrity is immediately detected when the agent reaches the next host and a data integrity check is performed on the agent.

b. Simulation Steps

- Enable attack scenario 1 from the property file (safer.properties)
- Start agent butler *m* (runm.bat)
- Start host *natalya* (runnat.bat)
- Run host *jinx* (agent roams from *jinx* to *natalya*) (run*jinx*.bat)
- When destination host *natalya* performs data integrity check on the agent, the compromise is discovered (see console of host *natalya*)

4.3.3 Signature Attack

a. Scenario Description

Signature attack refers to attacks that attempt to modify the digital signature on the agent data in order to cover any evidence of data compromise. To simulate this attack, one malicious host between *jinx* and *natalya* attempts to modify the digital signature from host *jinx*. It first modifies the data provided by *jinx* and somehow manages a valid signature. Despite being able to forge a valid signature, the new signature no longer matches the original signature chain (as the signatures are chained with each other). As a result, when the agent reaches the next host or returns to agent butler, the integrity validation will reveal the breakage of the signature chain, thus detecting the integrity compromise.

b. Simulation Steps

- Enable attack scenario 2 from the property file (safer.properties)
- Start agent butler *m* (runm.bat)
- Start host *natalya* (runnat.bat)
- Run host *jinx* (agent roams from *jinx* to *natalya*) (run*jinx*.bat)
- Deposit data on agent from *jinx* (runtest sg.edu.nus.safer.demo.DepositData <agent_host> <port> <agentID> <message>)
- Trigger agent return to butler *m* (runtest sg.edu.nus.safer.demo.AgentReturn <agent_host> <port> <agentID>)
- The agent butler who performs an integrity check on the returning agent detects the integrity compromise and throws an exception

4.3.4 Data Deletion Attack

a. Scenario Description

Another type of agent data compromise is in the form of data deletion. In this attack scenario, host *natalya* attempts to wipe out from agent *bond* all data collected from host *jinx*, including the itinerary record on *bond* showing its visit to *jinx*. Even if host *natalya* can somehow reconstruct a valid signature chain, once agent *bond* returns to the butler *m*, *m* will be able to detect the compromise when it compares the record itinerary against that stored in agent bond.

b. Simulation Steps

- Enable attack scenario 3 from the property file (safer.properties)

- Start agent butler *m* (runm.bat)
- Start host *natalya* (runnat.bat)
- Run host *jinx* (agent roams from *jinx* to *natalya*) (run*jinx*.bat)
- Deposit data on agent from *jinx* (runtest sg.edu.nus.safer.demo.DepositData <agent_host> <port> <agentID> <message>)
- Trigger agent return to butler *m* (runtest sg.edu.nus.safer.demo.AgentReturn <agent_host> <port> <agentID>)
- The agent butler who performs an itinerary verification on the returning agent detects the itinerary compromise and throws an exception

4.3.5 Key Seed Manipulation Attack

a. Scenario Description

Host *jinx* attempts to modify the encrypted dh parameter from *m* and replace it with another dh parameter. In order not to alert the next host *natalya* about the compromise, the new encrypted dh parameter must have a valid agent id (bond) and a valid host id (*natalya*) but different DH parameter. Once the agent reaches *natalya* and starts to communicate with the agent butler, the compromise will be detected as demonstrated. As agent roaming requires butler's directly involvement, the compromise will be definitely be detected.

b. Simulation Steps

- Enable attack scenario 4 from the property file (safer.properties)
- Start agent butler *m* (runm.bat)
- Start host *natalya* (runnat.bat)
- Run host *jinx* (agent roams from *jinx* to *natalya*) (run*jinx*.bat)
- Deposit data on agent from *jinx* (runtest sg.edu.nus.safer.demo.DepositData <agent_host> <port> <agentID> <message>)
- Trigger agent return to butler *m* (runtest sg.edu.nus.safer.demo.AgentReturn <agent_host> <port> <agentID>)
- The agent butler detects the itinerary compromise and throws an exception

4.3.6 Itinerary Attack

a. Scenario Description

In the earlier attack scenario simulation on data deletion, the agent itinerary is compromised along with data deletion. The roaming record to a particular host has been removed from agent itinerary. In this scenario, a malicious host attempts to fabricate an agent's roaming record by inserting additional itinerary entries into the agent. Host *natalya* modifies the agent itinerary during the agent's visit and dispatches the agent back to butler *m*. Again, the attack simulation ensures that the new signature chain is still valid. However, when the butler *m* receives the agent and performs an itinerary verification, the compromise can be detected.

b. Simulation Steps

- Enable attack scenario 5 from the property file (safer.properties)
- Start agent butler *m* (runm.bat)
- Start host *natalya* (runnat.bat)
- Run host *jinx* (agent roams from *jinx* to *natalya*) (run*jinx*.bat)
- Run echo (AB or BA) (runtest sg.edu.nus.safer.demo.SendEcho AB <agent_host> <port> <agentID> <message>)

Chapter 5

SAT

5.1 Overview

As SAT lays the foundation of SAFER security framework, the dependency of SAT is intentionally kept to a minimum. The only assumption of SAT is that entities should have a digital certificate and its corresponding private key. This implies that the entities in SAFER will be able to perform PKI operations using its private key. The main objective of this assumption is to enable entities to identify themselves to external parties easily.

The foundation of SAT begins with the message format used. Messages communicated during agent transport follow a general message format. The general message format includes features like timestamp, message sequence number, message digest etc. It is designed to protect the protocol against certain security attacks. One of these attack is replay attack, in which malicious host records a message and attempts to ‘play back’ at a later time. The use of timestamp, sequence numbering as well as message expiry in the general message format can effectively defend against replay attacks. The integrity of message is detected by the signed message digest attached. To provide most robust message digest, two message digest algorithms are used in cascade to ensure that the protocol will not be compromised even if one of the message digest algorithm is compromised.

With the general message format established, agent transport is comprised of three different transport protocols designed to address different security requirements. Supervised agent transport is the default transport protocol used in SAFER. It provides standard security feature that protects the agent against malicious host and allows the agent butler to have real-time information about the agent's whereabouts. However, the limitation of this protocol is the agent butler's active involvement in the transport process. In order to allow agent roaming while butler is offline, unsupervised agent transport is designed. This protocol does not require the active involvement of agent butler during roaming. The limitation is that information about agent roaming will reach the butler at a slightly later time as the acknowledgement messages of roaming are dispatched to the butler in asynchronous mode. In case the agent is somehow compromised, it may take slightly longer for the agent butler to realize than supervised agent transport. Both supervised and unsupervised agent transport are standard transport mechanisms in SAFER.

In order to allow applications with special needs to have customized transport mechanism and yet make use of SAFER framework, bootstrap transport protocol is proposed. In bootstrap transport protocol, a special transport agent carrying the customized transport protocol is first dispatched to the destination host under either supervised or unsupervised agent transport. The transport agent will perform the necessary authentication of the destination host before starting the actual agent transport using application-specific algorithms.

The general message format and three different agent transport algorithms will be explained in details in the next chapter.

5.2 Assumptions

As a prerequisite, each SAFER entity must carry a digital certificate issued by SAFER Certificate Authority, or SCA. In this way, agents, agent owners and hosts will all carry its own unique digital certificate. The certificate itself is used to establish the identity of a SAFER entity. The private component of the certificate has signing capability. This allows the certificate owner to authenticate itself to the SAFER community.

From the host's viewpoint, agent is a piece of foreign code that executes locally. In order to allow the host certain control over visiting agents, mobile agents executing in foreign host is not allowed to communicate directly with external parties. In other words, agents will not be able to establish socket connections directly with any entity outside the host, not even the agent's owner/butler. Agent receptionist will act as a middleman to facilitate and monitor agent communication with external party. There is an important reason that agent is not allowed to communicate directly with external parties. If the agent is able to communicate directly with external parties, it can easily 'smuggle' in other agents (could be malicious agents) into the host without informing the host. This defeats the purpose of agent transport. To make things worse, there is no way for the host to detect that the agent is bringing in other agents without proper authorization. As a result, the agent is not allowed to make direct communication with external parties in SAFER.

5.3 General Message Format

In SAFE, agent transport is achieved via a series of message exchanges. The format of each general message is as follows:

SAFER Message = Message Content + Timestamp + Sequence Number + MD(Message *Content + Timestamp + Sequence Number) + Signature(MD)* (7)

The main body of a SAFER message comprises of message content, a timestamp and a sequence number. The message content is defined by individual message. It can contain any information sent by the message issuer.

A timestamp is imposed on each message. It contains the issue time of the message and an expiry time of the same message. When a SAFER message reaches the recipient, the recipient should inspect the timestamp first. If the message arrives before the issue time of the message or after the expiry time of the message, the recipient should generate an alert to the message sender as well as the recipient's administrator. The duration between expiry time and issue time is set in the SAFER community. However, individual entities can choose to set a different duration based on their needs. The local setting should overwrite the general setting by SAFER. The general guideline is that the duration must be longer than the maximum tolerable time for message exchange to complete but slightly less than maximum tolerable agent transport time. The reason is that a message must remain valid when it reaches its recipient. Message transmission itself will incur certain time overhead. If the validity duration is shorter than the maximum tolerable time for message exchange, a message may become invalid by the time it reaches its recipient. However, to effectively prevent replay attack, the message should not have a lifetime longer than needed. When agent transport process completes, the message does not need to remain valid any more. As a result, the duration should be slightly shorter than the maximum tolerable agent transport time.

To further prevent replay attack, message exchanges between entities during agent transport is labeled according to each transport session. A running sequence number is included into the message body whenever a new message is exchanged. For example, if Alice sends a message to Virtual CD Mall with a sequence number 1, the next message sent by Alice to Virtual CD Mall will have a sequence number of 2. Similarly, messages from Virtual CD Mall to Alice will bear the same sequence number. In this way, if a message is lost during transmission or an additional message is received, the recipient will be able to detect it and alert both parties involved in the communication.

In order to protect the integrity of the main message body, a message digest is appended to the main message. The formula of the message digest is as follows:

$$\textit{Message Digest} = \textit{MD5}(\textit{SHA}(\textit{message\_body}) + \textit{message\_body}) \qquad (8)$$

The calculation of message digest is based on the most popular message digest algorithm, MD5 and SHA. By combining both algorithms, the formula leverages on the strength of both MD5 and SHA. Even if one of them is compromised, the overall security of the formula is still intact. If there is a transmission error in the message body, a recalculation of message digest based on the corrupted message body will generate a different set of message digest.

The message digest alone is not sufficient to protect the integrity of SAFER message. A malicious hacker can modify the message body and recalculate the value of message digest using the same formula and produce a seemingly valid message digest. To ensure the authenticity of the message, a digital signature on the message digest is generated for each SAFER message. Since only the message issuer has the private key to its digital certificate, no one else will be able to generate a valid signature based on a modified set of message digest. In addition to ensuring message integrity, the signature serves as a proof for non-repudiation as well.

If the message content is sensitive, it can be encrypted using a symmetric key algorithm (e.g., Triple DES). SAFER does not provide a general key exchange protocol for general messages. The secret key used for encryption will have to be decided in higher level (in different agent transport protocols).

The design of general message format covers quite a number of security concerns. The general security concerns (identification, authentication, secrecy, message integrity and non-repudiation) have been addressed. Some known security attacks to general messages (e.g. replay attack, man-in-the-middle attack) have been addressed. However, it is noted that the same attacks may be applied to the transport protocol itself. They will be further addressed in the design of different transport protocols.

To cater for different application concerns, two transport protocols were proposed in [2], supervised agent transport and unsupervised agent transport. However, it is realized that there will be additional security concerns for certain applications in which a proprietary transport protocol is desired. To cater for the needs of these applications, bootstrap agent transport protocol is designed. These three protocols will be discussed in the following sections in details.

5.4 Supervised Agent Transport

Supervised agent transport is designed for applications that require close supervision of agents. Under this protocol, agents have to request roaming permit from its owner or butler before it is able to start roaming. The owner has the option to deny the roaming request and prevent the agent from roaming to undesirable hosts. If Alice sends out a group of agents to search for the best bargain of Mariah Carey albums, one criteria is probably not to repeat searching the same CD shop. To meet the criteria, she can use supervised agent transport approach. Whenever an agent wishes to roam to a new online shop, Alice will search her list of visited shops to ensure that the new destination is not in the list. If the destination is already in the list, she will deny the roaming request and inform the agent to try some other shops, thus avoiding repeated effort. If the destination does not exist in the list, she can update the list with the new shop and signal the agent to go ahead. Without agent owner playing an active role in the transport protocol, it is difficult to have tight control over agent roaming.

The procedures of supervised agent transport is shown in Figure 7:

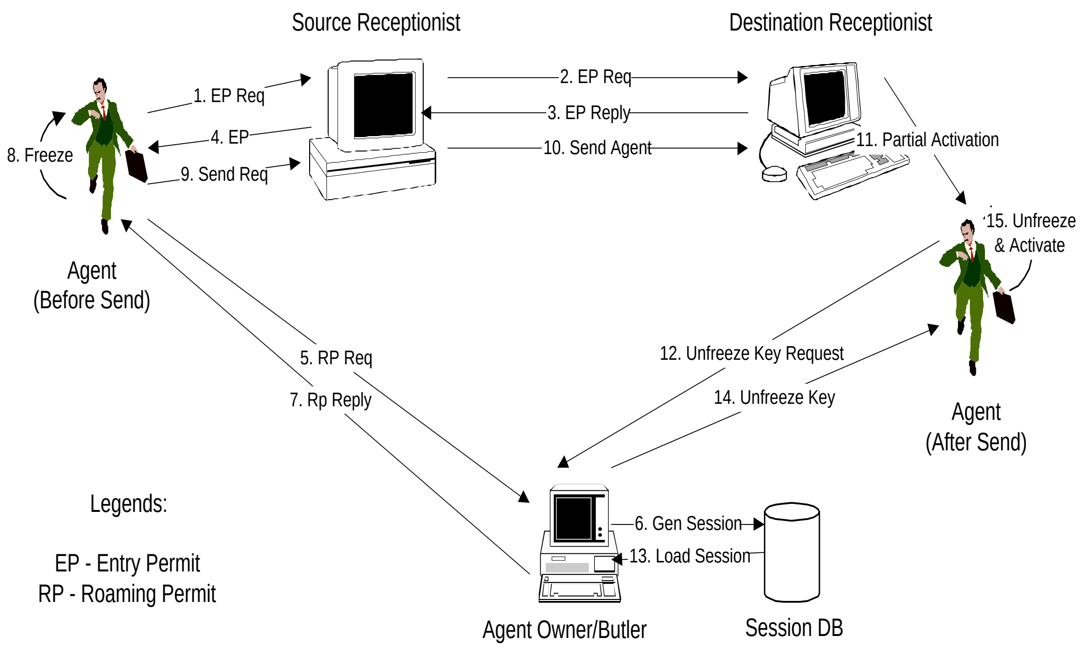


Figure 7 Supervised agent transport

5.4.1 Agent Receptionist

Agent receptionist is a process running at every host to facilitate agent transport. If an agent wishes to roam to a host, it should communicate with the agent receptionist at the destination host to complete the transport protocol. Every host will keep a pool of agent receptionist to service incoming agents. Whenever an agent roaming request arrives, an idle agent receptionist from the pool will be activated to entertain the request. In this way, a number of agents can be serviced concurrently. The number of agent receptionists in the pool should be set to the maximum number of acceptable concurrent visiting agents in the host. If the number of roaming requests exceeds the number of agent receptionists, the request will not be granted until some existing visiting agent leaves the host.

5.4.2 Request through source receptionist for entry permit

To initiate supervised agent transport, an agent needs to request for an entry permit from the destination receptionist. Unfortunately, the agent in a foreign host is not allowed to make direct communication with external parties due to security concerns. All communication between visiting agent and foreign parties (other agents outside the host, agent owner etc) goes through the agent receptionist. The request for an entry permit is first sent to the source receptionist. The request contains the requesting agent’s digital certificate and the destination address. The source receptionist will forward the agent’s digital certificate to the destination receptionist as specified in the agent’s request.

The destination receptionist can inspect the requesting agent’s information by reading its digital certificate and decide whether to issue entry permit based on its own authorization policy. If the destination receptionist decides to grant the request, an entry permit is generated and returned to the requesting agent. The entry permit will contain a random challenge, a serial number, a validity period and the digital certificate of the requesting agent and a digital signature by the destination receptionist on the entry permit.

The random challenge is used to authenticate the incoming agent. Its usage will be discussed later in the discussion of supervised agent transport protocol. For book keeping purpose, a serial number is included in the entry permit issued by a receptionist. This number should be unique to all entry permits issued by the same receptionist. It is possible that entry permits issued by different receptionists bear the same serial number. This will not cause any conflict in supervised agent transport. A timestamp is also part of the entry permit. Different from timestamps on general messages, the timestamp on entry permit specifies the validity of the entry permit. It is up to each receptionist to decide how long the issued entry permit remains valid. In order to prove the authenticity of the entry permit, the issuing receptionist needs to digitally sign the entry permit with its private key.

5.4.3 Request for roaming permit

Once the source receptionist receives the entry permit from destination receptionist, it simply forwards it to the requesting agent. The next step is for the agent to receive a roaming permit from its owner/butler. The agent sends the entry permit and address of its owner/butler to the source receptionist. Without processing, the source receptionist forwards the entry permit to the address as specified in the agent request.

The agent owner/butler can decide whether the roaming permit should be issued based on its own criteria. For example, the agent owner/butler can deny the roaming request if the agent is roaming to an undesirable site (repeated search or black-listed), grant the roaming permit if the destination is a ‘trusted’ host, or even consult the user if the roaming operation may cause significant impact to the user.

In any case, if the agent owner/butler decides to issue the roaming permit, it will have to generate a session number, a random challenge, a freeze/unfreeze key pair. The roaming permit should contain the session number, random challenge, freeze key, timestamp, entry permit and a signature on all the above from the agent owner/butler.

The session number is used to uniquely identify a roaming session. This number is stored in the session database together with other session information. When the agent reaches the destination and requests for the unfreeze key, session number will be used as a unique key to retrieve the corresponding unfreeze key.

In order to verify that the agent has indeed reached the intended destination, a random challenge is generated into the roaming permit. A digital signature on this random challenge is required for the destination to prove its authenticity. This will be discussed in greater detail later.

For the issuing of every roaming permit, a key pair is generated. A public key is included in the roaming permit for agents to encrypt or freeze its sensitive code/data during roaming. When the agent reaches the destination, it can obtain the private key (unfreeze key) from its owner to activate itself.

Same as entry permit, roaming permit also contains a timestamp that specifies the validity of the permit. As a general guideline, the validity should be the same as that in the entry permit unless the validity specified in the entry permit is deemed inappropriate.

Since a roaming permit is issued based on the entry permit presented, the entry permit will be part of the roaming permit. In this way, a roaming permit issued to entry permit A can not be used as a valid roaming permit to enable an agent roaming using entry permit B.

Finally, to provide non-repudiation, the agent owner/butler will digitally sign the roaming permit. Without a valid signature attached, a roaming permit will be void.

5.4.4 Agent Freeze

If the roaming request is granted, source receptionist will receive the roaming permit from the agent owner/butler and forward it to the requesting agent without processing. With the roaming permit and entry permit, the agent is now able to request for roaming from the source receptionist. In order to protect the agent during its roaming, sensitive function and codes inside the agent ‘body’ will be frozen. This is achieved using the freeze key in the roaming permit. Even if the agent is intercepted during its transmission, the agent’s capability is restricted. No much harm can be done to the agent owner/butler. To ensure a smooth roaming operation, the agent’s ‘life support systems’ cannot be frozen. Functions that are critical to the agent’s roaming capability, such as basic communication module, unfreeze operation module (which requires an unfreeze key to execute), must remain functional when the agent is roaming. All other functions and data not critical to agent roaming can be frozen and subsequently activated when the agent reaches its destination.

5.4.5 Agent Transport

Once frozen, the agent is ready for transmission over the Internet. To activate roaming, the agent sends a request containing the roaming permit to the source receptionist. The source receptionist can optionally verify the validity and authenticity of the roaming permit. Since the roaming permit (as well as the entry permit inside it) will be inspected one more time when it reaches the destination receptionist, the inspection by the source receptionist is optional.

If the agent’s roaming permit is valid, the source receptionist will transmit the frozen agent to the destination receptionist as specified in the entry permit. Once the transmission is completed, the source receptionist will terminate the execution of the original agent and make itself available to other incoming agents. The involvement of the source receptionist in the transport ends here.

5.4.6 Agent Pre-Activation

When the frozen agent reaches the destination receptionist, it will inspect the agent's roaming permit and the entry permit (contained in the roaming permit) carefully. By verifying the validity of both permits, the destination receptionist establish the followings:

a. The agent has been granted permission to enter the destination;

b. The entry permit carried by the agent has not expired;

c. The agent has obtained sufficient authorization from its owner/butler for roaming;

d. The roaming permit carried by the agent is not expired;

If the destination receptionist is satisfied with the agent's credentials, it will activate the agent partially and allows it to continue agent transport process.

5.4.7 Request for Unfreeze Key and Agent Activation

Although the agent has been activated, it is still unable to perform any operation since all sensitive codes/data is frozen. To unfreeze the agent, it has to request the unfreeze key from its owner/butler. To prove the authenticity of the destination, the destination receptionist is required to sign the random challenge in the roaming permit. The request for unfreeze key contains the session number, the certificate of destination and the signature on the random challenge.

The agent owner/butler can verify that the agent has indeed reached the right destination by validating the signature. If the signature is valid, the agent owner/butler will retrieve the unfreeze key based on session number, encrypt it using the destination's certificate and returns to the agent.

The destination receptionist can decrypt the unfreeze key using its private component of the certificate and passes the unfreeze key to the agent. Using the unfreeze key, the agent unfreezes itself. To prove to the destination host that the incoming agent is indeed the agent requesting the entry permit, the agent will use its private key to sign the random challenge in the entry permit and return to the destination receptionist. Once this signature has been verified, the destination receptionist fully activates the agent so that it can continue its execution in the new host.

The direct agent transport process is completed.

5.5 Unsupervised Agent Transport

Supervised agent protocol is not a perfect solution to agent transport. Although it provides tight supervision to an agent owner/butler, it has its limitations. Since the agent owner/butler is actively involved in the transport, the protocol inevitably incurs additional network traffic. This results in lower efficiency of the protocol. This especially significant when the agent owner/butler is located behind network with lower bandwidth, or the agent owner/butler is supervising a large number of agents. The network or the agent owner/butler can become the bottleneck and slow down the agent roaming process. For example, if Alice is behind an unstable network, she may not want to supervise her agents' roaming, especially if the agents are carrying out non-sensitive, independent tasks.

The steps involved in unsupervised agent transport is shown in Figure 8 below:

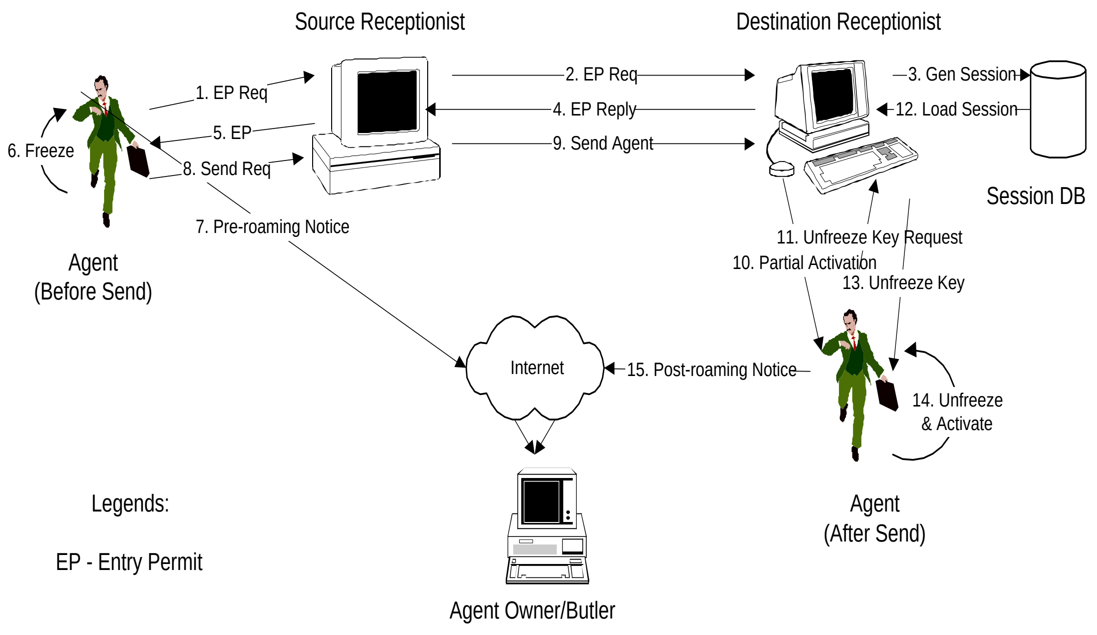


Figure 8 Unsupervised agent transport

5.5.1 Request for Entry Permit

Same as supervised agent transport, unsupervised agent transport is initiated by an agent requesting an entry permit from a destination receptionist through a source receptionist. The request comprises of the agent's certificate, a random challenge, the address of the destination receptionist. The destination receptionist will use the agent's certificate to identify the requesting agent. The random challenge will be used in later stages to authenticate the destination receptionist. When the source receptionist receives the request, it will pass this to the destination receptionist (as specified in the request) without much processing.

On receiving the entry request, the destination receptionist will have to decide whether the request should be granted. Same as supervised agent transport, each destination receptionist can have its own criteria on granting entry permits. If the destination receptionist decides to issue an entry permit, it will generate a serial number. This number must be unique within the destination receptionist and will be used to identify the transport session in later stages. At the same time, a freeze/unfreeze key pair is generated. The freeze key will be sent to the agent while the unfreeze key is stored in database as part of the session information. To challenge the authenticity of the requesting agent, a random challenge is generated and included in the entry permit. Similar to the entry permit in supervised agent transport, the validity period and the requesting agent's certificate are included into the entry permit. Finally, the destination receptionist signs the entry permit and sends it to the source receptionist. The source receptionist simply forwards the permit to the requesting agent.

5.5.2 Pre-roaming Notification

Unlike supervised agent transport, the agent does not need to seek an explicit approval to roam from its owner/butler. Instead, a pre-roaming notification is sent to the agent owner/butler through indirect means. Typically the notification can be in the form of email or UDP package. This notification contains the address information of source and destination receptionist only. It serves to inform the agent owner/butler that the agent has started its roaming. Since the notification is indirect, the agent does not need to wait for the owner/butler's reply before roaming.

5.5.3 Agent Freeze

Similar to supervised agent transport, the agent has to freeze its sensitive data/function before roaming. In supervised agent transport, the freeze key comes from the agent owner/butler. Since agent owner/butler is no longer directly involved in unsupervised agent transport, the agent will use the freeze key generated by the destination receptionist instead.

5.5.4 Agent Transport

Once the agent freezes its sensitive data/function, it will make a request to the source receptionist for the actual transport. The source receptionist will send the frozen agent to the destination receptionist as specified in the entry permit. Once the transport is completed, the source agent receptionist terminates the execution of the original agent and gets ready to service another incoming agent.

5.5.5 Request for Unfreeze Key

When the destination receptionist receives the agent, it will first inspect the agent's entry permit. If the entry permit is authentic and valid, it will activate the agent partially so that it can complete the roaming. Once the agent is activated, it will first request the unfreeze key from destination receptionist using the serial number in the entry permit. As communication between the agent and destination receptionist takes place locally, there is no need to encrypt the communication. The destination receptionist can fetch the unfreeze key from its session depository and return it to the agent. With the unfreeze key, the agent is able to unfreeze its sensitive data/function.

5.5.6 Agent Activation

To request for full activation, the agent must sign the random challenge in the entry permit and send it to the destination receptionist. If the signature is verified successfully, the destination receptionist can be convinced that the agent arrived is indeed the agent requesting for entry permit. To authenticate itself, the destination receptionist will sign the random challenge in the entry permit request and passes it to the agent. If this signature is verified, the agent is convinced that it has reached the right destination. Once mutual authentication is completed, the destination receptionist will fully activate the agent.

5.5.7 Post-roaming Notification

On full activation, the agent must send a post-roaming notification to its owner/butler. This will inform the agent owner/butler that the agent roaming has bee completed successfully. Again, this notification will take place through an indirect channel so that the agent does not need to wait for any reply before continuing with its normal execution.

The unsupervised agent transport is completed when the post-roam notification has been sent out.

5.6 Bootstrap Agent Transport

Both supervised and unsupervised agent transport make use of a fixed protocol for agent transports. The procedures for agent transport in these two protocols have been clearly defined without much room for variations. It is realized that there exist applications that require special transport mechanism for their agents. For example, applications that involve highly sensitive content may wish to use a proprietary protocol for their agent transports. In order to allow this flexibility, SAFER provides a third transport protocol, bootstrap agent transport. Under bootstrap agent transport, agent transport is completed in two phases. The first phase is to send a transport agent to the destination using either supervised or unsupervised agent transport. In the second phase, the transport agent takes over the role of destination receptionist and continues the transport of its parent agent with its own agent transport protocols. In this way, different applications can implement their transport agents using the preferred transport mechanisms and still be able to make use of the SAFER agent transport. Bootstrap agent transport is illustrated in Figure 9 below:

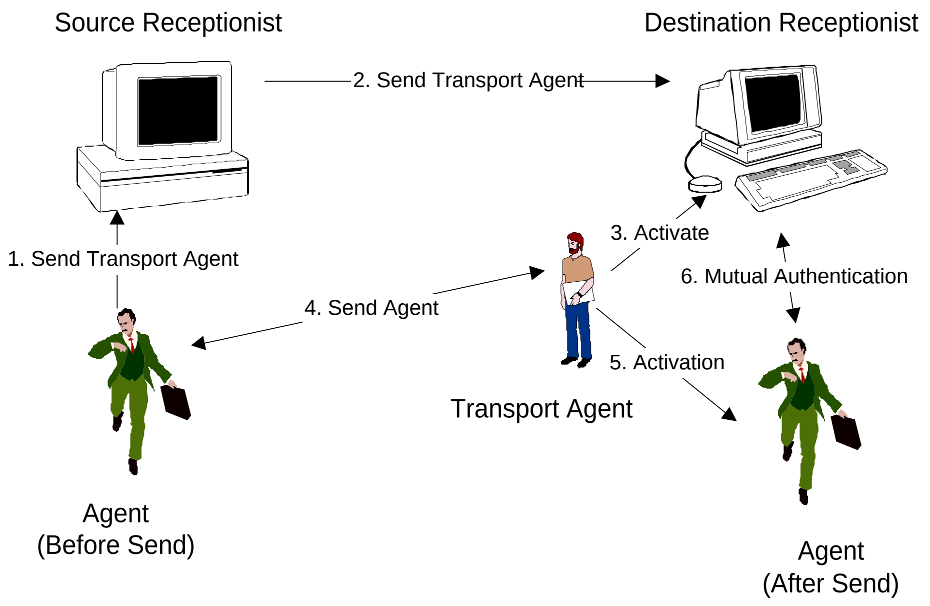


Figure 9 Bootstrap agent transport

In the first phase, the transport agent is sent to the destination receptionist using either supervised or unsupervised agent transport with some modifications. The original supervised and unsupervised agent transport requires agent authentication and destination authentication to make sure that the right agent reaches the right destination. Under bootstrap agent transport, the transmission of transport agent does not require both agent authentication and destination authentication.

Once the transport agent reaches the destination, it starts execution in a restricted environment. It is not given the full privilege as a normal agent does because of the fact that it has yet to authenticate itself to the destination. Under the restricted environment, the transport agent is not allowed to interact with local host services. It is only allowed to communicate with its parent until the parent reaches destination. A maximum time frame is imposed on the transport agent during which the transmission of its parent must complete. This is to prevent the transport agent from hacking attempts to local host. SAFER allows individual transport agents be customized to use any secure protocol for parent agent transmission. Concerns such as anonymity, secrecy, integrity etc should be taken care of by the transport agents. If the algorithm used by the transport agent is not secure, the whole agent may be compromised. In SAFER, parent agent is assumed the responsibility to make sure its transport agent uses a secure transport protocol.

When the parent agent reaches the destination, it can continue the handshake with the destination receptionist and perform mutual authentication directly. The authentication scheme is similar to that in supervised/unsupervised agent transport.

Chapter 6

SAT Analysis and Prototype

6.1 Implementation

To prototype the design of agent transport, the protocol discussed above is implemented.

The prototype is built on Windows 95/NT platform using Java. Since Java is a platform-independent language, the prototype can be deployed to any other platform that supports JVM (Java Virtual Machine). There are a few reasons why Java is chosen as the implementation language. Firstly, the most powerful feature of Java – platform independence - makes it the ideal language for building Internet-based applications. Internet is an architecture that is made up of a vast variety of platforms. In order to provide interoperability across multi-vendor platforms, a truly platform independent language is desired. With Java, the prototype can be build once and run anywhere on other platforms.

Further more, the garbage collector feature of Java significantly reduces the programming effort and allows developers to concentrate on programming logic rather than taking care of memory. Unlike some other languages such as C/C++, Java VM manages memory automatically through its garbage collector. Memory of unreferenced objects will be automatically reclaimed when the garbage collector is activated, thus reducing memory consumption to the minimum.

Another feature that benefits the prototyping is thread-safe. Java language makes it easy to develop multi-thread applications. Threading is taken care of by JVM so that applications using Java threading is automatically thread-safe. In other languages, extra effort is needed to ensure the program runs normally under multi-thread scenario.

As a first step in the prototyping, unsupervised agent transport has been implemented. Two agent receptionists are setup in different hosts simulating the source host and destination host. An agent carrying certain functions is invoked from the source host. It kicks off a series of message exchanges under unsupervised agent transport and eventually reaches the destination host. During the process, source receptionist and destination receptionist are involved in the handshake. When the agent reaches the destination, it successfully unfreezes itself and is activated for normal execution. During the simulation, two indirect messages are sent to the agent owner/butler (pre-roaming notice and post-roaming notice) as stipulated in the unsupervised agent transport protocol.

Functions carried by the agents are loaded into the agent body before roaming operation. They will be preserved throughout the agent transport. All functions carried are classified into sensitive functions and non-sensitive functions. Examples of sensitive functions are digital signature generation, negotiation strategy, mission statement, etc. Sensitive functions will be encrypted during the actual transmission. Non-sensitive functions refer to both functions with less sensitivity and functions that are vital to agent transport. Functions with less sensitivity do not need to be encrypted, and functions that are vital to agent transport cannot be encrypted or agent transport will not be able to perform regularly.

In the implementation, encryption on agent functions is done by first converting the agent function's byte-code into a binary stream (using the serialization feature of Java), and subsequently perform symmetric key encryption on the binary stream. The encrypted byte stream is carried in the agent body during agent transmission. When the agent reaches the destination, the encrypted byte stream will be decrypted into the original binary stream. From the original byte stream, the byte-code can be reconstructed and the agent function class dynamically loaded. The serialization feature of Java significantly reduces the programming complexity here.

The flow of unsupervised agent transport protocol implementation is summarized as follows:

6.1.1 Entry Permit Request (Agent to Source Receptionist)

Message content: agent certificate, destination address, and purpose of visit description.

6.1.2 Entry Permit Request (Source Receptionist to Destination Receptionist)

Message content: agent certificate, purpose of visit description.

6.1.3 Session Generation (Destination Receptionist)

Action: Generate random session key, generate random challenge, generate freeze/unfreeze key pair, and store session information to local database.

6.1.4 Issue Entry Permit (Destination Receptionist to Source Receptionist)

Message content: agent description and entry permit (content of entry permit is discussed in the earlier section).

6.1.5 Entry Permit Reply (Source Receptionist to Agent)

Message content: entry permit.

6.1.6 Agent Freeze (Agent)

Action: generate random session key, encrypt sensitive functions with session key, encrypt session key with freeze key.

6.1.7 Pre-Roaming Notice (Agent to Agent Owner/Butler – Indirect)

The notice is sent as an email message with destination address in the message body.

6.1.8 Send Request (Agent to Source Receptionist)

Message content: entry permit, destination address and encrypted agent.

6.1.9 Send Agent (Source Receptionist to Destination Receptionist)

Message content: entry permit and encrypted agent.

6.1.10 Partial Activation (Destination Receptionist to Agent)

Action: activate the agent for execution to continue the agent transport process.

6.1.11 Unfreeze Key Request (Agent to Destination Receptionist)

Message content: agent certificate, entry permit, and session identifier.

6.1.12 Load Session (Destination Receptionist)

Action: validate entry permit, load unfreeze key from database based on session identifier.

6.1.13 Unfreeze Key Reply (Destination Receptionist to Agent)

Message content: unfreeze key.

6.1.14 Unfreeze and Activation (Agent)

Action: decrypt session key using unfreeze key, and decrypt sensitive functions using the session key.

6.1.15 Post-Roaming Notice (Agent to Agent Owner/Butler – Indirect)

An email message is sent out to agent owner/butler notifying the success of agent transport.

Chapter 7

Conclusion

In the first part of the thesis, a new data integrity protection protocol, SADIS, is proposed under the SAFER research initiative. Besides being secure against a variety of attacks and robust against vulnerabilities of related work in the literature, the research of SADIS includes the objective of efficiency. This is reflected in the minimized use of PKI operations and reduced message exchanges between the agent and the butler and results in superior performance compared with other researches in the literature. The introduction of variation to DH key exchange and evolving communication session key further strengthened the security of the design. Unlike some existing literature, the data integrity protection protocol aims not only to detect data integrity compromise, but more importantly, to identify the malicious host.

In the second part of the thesis, the security of agent transport is addressed in SAT. SAT is the foundation of SAFER framework to provide mobile agents with roaming capability without compromising security. General security concerns as well as security concerns raised by agent transport have been carefully addressed. The design of the protocol also takes into consideration different concerns for different applications. Instead of standardizing on one transport protocol, three different transport protocols are designed, catering to various needs. Based on the level of control desired, one can choose between supervised agent transport and unsupervised agent transport. For applications that require high level of security during agent roaming, bootstrap agent transport is provided so that individual applications can customize their transport protocols.

With security, efficiency, and flexibility as its main design focuses, SADIS and SAT work with other security mechanisms under SAFER (e.g., SAFER Certification, Agent Battery – A special mechanism to measure the risk factors accumulated for a roaming session) to provide mobile agents with a secure platform. Special considerations were taken during the design of both protocols to ensure an efficient and practical solution. This can be demonstrated in the benchmark of the prototype.

Besides agent data integrity and agent transport security, there are other security concerns to be addressed in SAFER. One such concern is a mechanism to assess the agent’s accumulated risk level as it roams. There have been some considerations for using ‘agent battery’ concept to address this during the earlier stages of the research. However, the research in this area is independent of this thesis and requires separate consideration. Furthermore, in order to establish the identity of different agents from different agent communities, certain level of certification (e.g., SAFER certification) or agent passport is required. More research can be conducted in this area following the earlier exploration in this research.

References

[1] Fangming Zhu, Sheng-Uei Guan, Yang Yang, C.C. Ko, "SAFER E-Commerce: Secure Agent Fabrication, Evolution and Roaming for E-Commerce". Accepted for publication in Electronic Commerce: Opportunities and Challenges, IDEA Group Publishing, USA, 2000.

[2] Sheng-Uei Guan, Yang Yang, "SAFE: Secure-roaming Agent For E-commerce", Computer & Industrial Engineering Conference'99, Melbourne, Australia, pp33-37, 1999.

[3] Sheng-Uei Guan, Yang Yang, "SAFE: Secure Agent roaming for E-Commerce", Computer & Industrial Engineering Journal, 2002.

[4] Yang Yang, Sheng-Uei Guan, "Intelligent Mobile Agents for E-Commerce: Security Issues and Agent Transport", Electronic Commerce: Opportunities and Challenges, Idea Group Publishing, USA, 2000.

[5] Hock Boon Chionh, Sheng-Uei Guan and Yang Yang, "Ensuring the Protection of Mobile Agent Integrity: The Design of an Agent Monitoring Protocol", Proceedings,

IASTED International Conference on Advances in Communications (AIC 2001), Rhodes, Greece, pp96-99, July 2001.

[6] Tianhan Wang, Sheng-Uei Guan and Tai Khoon Chan, “Integrity Protection for Code-on-Demand Mobile Agents in E-Commerce”, Journal of Systems and Software, pp211-221, Vol. 60, Iss. 3, 2002.

[7] Volker Roth, “On the robustness of some Cryptographic Protocols for Mobile Agent Protection”, Mobile Agents 2001 (MA’01), pp1-14, 2001.

[8] Anand R. Tripathi and others, “Design of the Ajanta system for mobile agent programming”, Journal of Systems and Software, Vol. 62, Iss. 2, pp123-140, 2002.

[9] G. Karjoth, N. Asokan and C. Gulcu, “Protecting the computation results of free-roaming agents”, Mobile Agents, 1998.

[10] Walter Binder, Volker Roth, “Secure mobile agent systems using Java, where are we heading”, ACM Symposium on Applied Computing (SAC), pp115-119, 2002.

[11] A. Corradi, M. Cremonini, R. Montanari, and C. Stefanelli, “Mobile Agents and Security: Protocols for Integrity”, Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS’99), 1999.

[12] P. Bellavista, A.Corradi, and C. Stefanelli, "Protection and Interoperability for Mobile Agents: a Secure and Open Programming Environment", IEICE Transactions on Communications, 2000.

[13] Jong-Youl Park, Dong-Ik Lee, and Hyung-Hyo Lee, "One-Time Key Generation System for Agent Data Protection", IEICE Transactions on Information and Systems, pp535-545, 2002.

[14] Johansen, D., Lauvset, K. J., Renesse. R, Schneider, F. B., Sudmann, N.P., and Jacobsen, K., "A Tacoma Retrospective", Software – Practice and Experience, pp605-619, 2002.

[15] Bruce Schneier. "Applied Cryptography: Protocols, Algorithms, and Source Code in C", 2nd Ed., John Wiley & Sons, Inc, New York, 1996.

[16] N. Borselius, "Mobile Agent Security", Electronics & Communication Engineering Journal, Vol. 14, no 5, IEE, London, UK, pp211-218, 2002.

[17] Andrew S. Patrick, "Building Trusthworthy Software Agents", IEEE Journal of Internet Computing, pp46 – 53, 2002.

[18] Niklas Borselius, Namhyun Hur, Marek Kaprynski and Chris J. Mitchell. "A security architecture for agent-based mobile systems", In Proceedings of the Third

International Conference on Mobile Communications Technologies – 3G2002, London, UK, IEE Conference Publication 489, pp312–318, 2002.

[19] N. Borselius, C.J. Mitchell and A.T. Wilson. "On mobile agent based transactions in moderately hostile environments". In B. De Decker, F. Piessens, J. Smits and E. Van Herreweghen, editors, Advances in Network and Distributed Systems Security - Proceedings of the IFIP TC11 WG11.4 First Annual Working Conference on Network Security, Kluwer Academic Publishers, Boston, pp173-186. 2001.

[20] Stan Franklin, etc. "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[21] D. Johansen, K. Marzullo, and K.J. Lauvset, "An Approach towards an Agent Computing Environment", ICDCS'99 Workshop on Middleware, 1999.

[22] F. B. Schneider, "Towards Fault-tolerant and Secure Agentry", Invited paper, 11th International Workshop on Distributed Algorithms, Saarbrücken, Germany, 1997.

[23] T. Finin, J. Weber, "Draft specification of the KQML agent communication language", et al., http://www.cs.umbc.edu/kqml/kqmlspec/spec.html, 1993.

[24] C. Thirunavukkarasu, T. Finin, and J. Mayfield, “Secret Agents – A Security Architecture for the KQML Agent Communication Language”, CIKM’95 Intelligent Information Agents Workshop, Baltimore, 1995.

[25] D. Rus, R. Gray, and D. Kotz, “Transportable information agents”. In Michael Huhns and Munindar Singh, editors, Readings in Agents. Morgan Kaufmann Publishers, San Francisco, 1997.

[26] D. E. White, “A comparison of mobile agent migration mechanisms”, Senior Honors Thesis, Dartmouth College, 1998.

[27] D. Rus, R. Gray, and D. Kotz, “Autonomous and adaptive agents that gather information”, AAAI’96 International Workshop on Intelligent Adaptive Agents, 1996.

[28] R. Gray, “Agent TCL: A flexible and secure mobile-agent system”, Ph. D. thesis, Department of Computer Science, Dartmouth College, 1997.

[29] B. Schneier, “Applied Cryptography: Protocols, Algorithms, and Source Code in C”, 2nd Ed., John Wiley & Sons, Inc, New York, 1996.

[30] C. Guilfoyle, “Intelligent Agents: The new revolution in software”, OVUM, London, 1994.

[31] S. Corley, “The application of Intelligent and Mobile Agents to Network and Service Management”, et al., 5th International Conferences on Intelligence in Services and Networks, IS&N’98, Antwerp, Belgium, May 1998 Proceedings, 1995.

[32] T. Finin, Fritzon R., McKay D., McEntire R., “KQML – A Language Protocol for Knowledge and Information Exchange”, CS Tech. Report CS-94-02, University of Maryland, 1994.

[33] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko, “Agent TCL: targeting the needs of mobile computers”, IEEE Internet Computing, Vol. 1, No. 4, pp58 – 67, 1997.

[34] R. Schoonderwoerd, O. Holland, and J. Bruten, “Ant-like agents for load balancing in telecommunications networks”, Proceedings of the 1997 1st International Conference on Autonomous Agents, Marina Del Rey, California, U.S.A., pp209 – 216, 1997.

[35] J. B. Odubiyi, D. J. Kocur, S. M. Weinstein, N. Wakim, S. Srivastava, C. Gokey, and J. Graham, “SAIRE – A Scalable Agent-Based Information Retrieval Engine”, Proceedings of the Autonomous Agents 97 Conference, Marina Del Rey, California, U.S.A., pp292 – 299, 1997.

[36] Alan O. Frier, Philip Karlton, Paul C. Kocher, “The SSL Protocol Version 3.0”, http://wp.netscape.com/eng/ssl3/draft302.txt, 1996.

Appendix A

Major Source Code for SADIS

Source code of Agent.java:

```
package sg.edu.nus.safer.object;

import java.util.*;
import sg.edu.nus.base.crypto.CryptoFactory;
import sg.edu.nus.base.crypto.IMDService;
import sg.edu.nus.base.crypto.IKeyExchange;
import sg.edu.nus.base.crypto.IEncryptionService;
import sg.edu.nus.base.crypto.RSAUtils;
import sg.edu.nus.base.exception.BaseException;
import sg.edu.nus.base.logger.DefaultLogger;
import sg.edu.nus.base.util.ObjectUtil;
import sg.edu.nus.base.util.HexUtil;
import sg.edu.nus.base.util.PauseUtil;
import sg.edu.nus.safer.constant.SaferConst;
import sg.edu.nus.safer.hcp.HCP10Client;
import sg.edu.nus.safer.knp.KNP10Client;
import sg.edu.nus.safer.knp.KNPHelloMessage;
import sg.edu.nus.safer.knp.KNPReplyMessage;
import sg.edu.nus.safer.cert.CertDB;
import sg.edu.nus.safer.attack.AttackUtils;
import com.integrosys.base.techinfra.propertyfile.PropertyManager;
import crypto.MasterCert;
import firefox.crypto.CryptoUtils;
import firefox.crypto.RSAPublicKey;

public class Agent extends object.Agent {
    private byte[] m_key_seed =
PropertyManager.getValue("agent.InitialKeySeed",
"supermanmagicgal").getBytes();
    private int m_skey_counter = 0;
    private String m_currentHostID;
    private Host m_currentHost;
    private IKeyExchange m_keyExchange;
    private byte[] m_encrypted_y;
    private AgentSuitcase m_agentSuitcase;

      // Constructor
    public Agent(String ID, String description, MasterCert cert,
String ownerID, String ownerEmail) {
        super(ID, description, cert, ownerID, ownerEmail);
        init();
    }

      // Initializes the agent
    private int init() {
        m_agentSuitcase = new AgentSuitcase(getID());
        return 1;
    }

    public void setKeySeed(byte[] key_seed) {
        m_key_seed = key_seed;
        m_skey_counter = 0;
    }

    private byte[] getKeySeed() {
        return m_key_seed;
    }

      // Get the next skey counter
    private int nextCounterValue() {
        DefaultLogger.info(this, "Getting next counter value,
increasing by one from " + m_skey_counter);
        return ++m_skey_counter;
    }

      // Decrease skey counter (used for attack simulation)
    public void decreaseCounterValue() {
        DefaultLogger.info(this, "Decreasing counter value, this
should be called only if it is necessary to rollback the counter
value in case of failure!");
        m_skey_counter--;
    }

      // Obtain the next communication session key
    public byte[] getSessionKey(String host_id) throws BaseException
{

        IMDService md = CryptoFactory.getMDService();
        md.initialize();
        if (getKeySeed() != null) md.update(getKeySeed());
        if (host_id != null) md.update(host_id);
        md.update(new Integer(nextCounterValue()).toString());
        byte[] hash = md.digest();
        byte[] key = new byte[SaferConst.SKEY_LEN];
        System.arraycopy(hash, 0, key, 0, SaferConst.SKEY_LEN);
        return key;
    }

      // Get data encryption key
    public byte[] getDEK(String host_id) throws BaseException {
        IMDService md = CryptoFactory.getMDService();
        md.initialize();
        if (getKeySeed() != null) md.update(getKeySeed());
        if (host_id != null) md.update(host_id);
        byte[] hash = md.digest();
        byte[] key = new byte[SaferConst.DEK_LEN];
        System.arraycopy(hash, 0, key, 0, SaferConst.DEK_LEN);
        return key;
    }

    public String getButlerHost() {
        return PropertyManager.getValue("agent.Butler.Address");
    }
    public int getButlerPort() {
        return PropertyManager.getInt("agent.Butler.Port");
    }
    public void setCurrentHostID(String hostID) {
        m_currentHostID = hostID;
    }
    public String getCurrentHostID() {
        return m_currentHostID;
    }

    public void setCurrentHost(Host host) { m_currentHost = host; }
    public Host getCurrentHost() { return m_currentHost; }

       // Process echo message from agent butler
    public void processEcho(byte[] cipher) throws BaseException {
        DefaultLogger.info(this, "Processing echo message");
        byte[] plain = null;
        if (SaferConst.USE_SAFER) {
            IEncryptionService encryptor =
CryptoFactory.getEncryptionService();
            plain =
encryptor.decrypt(getSessionKey(getCurrentHostID()), cipher);
            plain = CryptoUtils.unpad(plain);
        } else {
            plain = cipher;
        }
        DefaultLogger.info(this, "Butler says: " + new String(plain));
    }

      // Send echo message to agent butler
    public void sendEchoToButler(String message) throws BaseException
{
        DefaultLogger.info(this, "Send echo message to butler");
        byte[] cipher = null;
        if (SaferConst.USE_SAFER) {
            IEncryptionService encryptor =
CryptoFactory.getEncryptionService();
            byte[] plain = CryptoUtils.pad(message.getBytes());
            cipher =
encryptor.encrypt(getSessionKey(getCurrentHostID()), plain);
        } else {
            cipher = message.getBytes();
        }
        KNP10Client client = new KNP10Client(getButlerHost(),
getButlerPort());
        client.sendABEcho(getID(), getCurrentHostID(), cipher);
    }

      // Prepares hello message (with SAFER)
    private KNPHelloMessage prepareHelloMessageWithSAFER(String
hostID, String hostAddr, int hostPort) throws BaseException {
        DefaultLogger.info(this, "Start to roam to host with
SAFER ...");
        // initialize DH key exchange
        DefaultLogger.info(this, "Generating DH private and public
parameter ...");
        m_keyExchange = CryptoFactory.getKeyExchange();
        m_keyExchange.initialize();

        // prepare plain information to send to butler
        // including target host ID, dh public param
        DefaultLogger.info(this, "Preparing target host ID, dh public
param for key exchange ...");
        Object dh_public = m_keyExchange.getPublicKey();
        //String targetHostID =
PropertyManager.getValue("targethost.ID");
        String targetHostID = hostID;
        //String host =
PropertyManager.getValue("targethost.Address");
        //int port = PropertyManager.getInt("targethost.Port");
        String host = hostAddr;
        int port = hostPort;
        Vector v = new Vector(3);
        v.addElement(targetHostID);
        v.addElement(dh_public);
        v.addElement(getID());



        // serialize data carrier v in order to encrypt
        byte[] cipher = null;
        try {
            byte[] plain = ObjectUtil.objectToBytes(v);
            DefaultLogger.info(this, "Plain data size is: " +
plain.length);
            plain = CryptoUtils.pad(plain);
            IEncryptionService encryptor =
CryptoFactory.getEncryptionService();
            DefaultLogger.info(this, "Plain before encryption is: " +
HexUtil.bytesToHex(plain));
            cipher =
encryptor.encrypt(getSessionKey(getCurrentHostID()), plain);
            DefaultLogger.info(this, "Encrypted result is (size " +
cipher.length + "): " + HexUtil.bytesToHex(cipher));
        } catch (Throwable t) {
            t.printStackTrace();
            throw new BaseException("Exception caught in Agent while
encrypting information with session key");
        }

        KNPHelloMessage message = new KNPHelloMessage();
        message.addData(getCurrentHostID());
        message.addData(cipher);
        message.addData(getID());
        message.addData(host);
        message.addData(new Integer(port));
        message.addData(targetHostID);
        return message;
    }

      // Prepares hello message (not using SAFER, for comparison
purpose)
    private KNPHelloMessage prepareHelloMessageWithoutSAFER(String
hostID, String hostAddr, int hostPort) throws BaseException {
        DefaultLogger.info(this, "Start to roam to host without
SAFER ...");
        // prepare plain information to send to butler
        // including target host ID but no dh public param
        DefaultLogger.info(this, "Preparing target host ID for
roaming ...");
        //String targetHostID =
PropertyManager.getValue("targethost.ID");
        String targetHostID = hostID;
        //String host =
PropertyManager.getValue("targethost.Address");
        String host = hostAddr;
        //int port = PropertyManager.getInt("targethost.Port");
        int port = hostPort;
        KNPHelloMessage message = new KNPHelloMessage();
        message.addData(targetHostID);
        message.addData(getID());
        message.addData(host);
        message.addData(new Integer(port));
        return message;
    }
      // Trigger agent roaming to next host
    public void roamToHost(String hostID, String hostAddr, int
hostPort) throws BaseException {
        KNPHelloMessage message = null;
        if (SaferConst.USE_SAFER) message =
prepareHelloMessageWithSAFER(hostID, hostAddr, hostPort);
        else message = prepareHelloMessageWithoutSAFER(hostID,
hostAddr, hostPort);
        // send message to butler
        PauseUtil.pause();
        KNP10Client client = new KNP10Client(getButlerHost(),
getButlerPort());
        client.connectToServer();
        KNPReplyMessage reply = client.sendHelloMessage(message);
        if (SaferConst.USE_SAFER) processReplyMessageWithSAFER(reply);
        else processReplyMessageWithoutSAFER(reply);
        DefaultLogger.info(this, "About to request current host to
send itself to target host ...");
        PauseUtil.pause();
        //String host =
PropertyManager.getValue("targethost.Address");
        //int port = PropertyManager.getInt("targethost.Port");
        getCurrentHost().sendAgent(this, hostAddr, hostPort);
    }

      // process KNP reply message (using SAFER)
    private void processReplyMessageWithSAFER(KNPReplyMessage reply)
throws BaseException {
        DefaultLogger.info(this, "Processing reply message with
SAFER ...");
        byte[] cipher = (byte[]) reply.getDataAt(0);

        IEncryptionService encryptor =
CryptoFactory.getEncryptionService();
        byte[] plain =
encryptor.decrypt(getSessionKey(getCurrentHostID()), cipher);
        plain = CryptoUtils.unpad(plain);
        DefaultLogger.info(this, "Decrypted plain length is: " +
plain.length);

        m_encrypted_y = plain;
        if (AttackUtils.isHostAttackEnabled(4,
getCurrentHost().getID())) {
            // use a dummy message to replace encrypted y
            DefaultLogger.info(this, "creating fake DH param");
            m_keyExchange = CryptoFactory.getKeyExchange();
            m_keyExchange.initialize();
            Vector v = new Vector();
            v.addElement(m_keyExchange.getPublicKey()); // DH param
            v.addElement("natalya");    // dummy host ID, use the
```

```
right host
            v.addElement("bond");       // dummy agent ID
            v.addElement(new Date());   // dummy timestamp
            plain = ObjectUtil.objectToBytes(v);
            // Encrypt plain with destination host's public key
            RSAPublicKey host_pub_key = CertDB.getCert("natalya");
            m_encrypted_y = RSAUtils.publicEncrypt(plain,
host_pub_key);
        }

        DefaultLogger.info(this, "Persisted encrypted y, proceed to
roam to destination");

        // hack
        //continueKeyNegotiationWithNewHost();
    }

      // Continue KNP with the next host
    private void continueKeyNegotiationWithNewHost() throws
BaseException {
        if (!SaferConst.USE_SAFER) return;
        DefaultLogger.info(this, "Trying to reestablish key seed
after reaching new host");

        // request host to decrypt
        Object dh_public_key =
getCurrentHost().decryptDHPublicKey(m_encrypted_y);
        if (!SaferConst.USE_NEW_KNP_FORMULA) {
            // no additional processing required
        } else {
            DefaultLogger.info(this, "Using new KNP formula");
            Vector v = (Vector) dh_public_key;
            dh_public_key = v.elementAt(0);
            String intendedHostID = (String) v.elementAt(1);
            String intendedAgentID = (String) v.elementAt(2);
            // discard timestamp, no need to check
            // check if host ID matches
            if (!getCurrentHost().getID().equals(intendedHostID)) {
                throw new BaseException("Host ID mismatch!");
            }
            // check if agent ID matches
            if (!getID().equals(intendedAgentID)) {
                throw new BaseException("Agent ID mismatch!");
            }
        }
        byte[] key_seed = m_keyExchange.negotiate(dh_public_key);
        setKeySeed(key_seed);
        DefaultLogger.info(this, "New Key Seed: " +
HexUtil.bytesToHex(key_seed));
    }
      // Process KNP reply message (not using SAFER)
    private void processReplyMessageWithoutSAFER(KNPReplyMessage
reply) throws BaseException {
        DefaultLogger.info(this, "Processing reply message without
SAFER ...");
        String msg = (String) reply.getDataAt(0);
        DefaultLogger.info(this, "Butler message: " + msg);
        DefaultLogger.info(this, "Proceed to roam to destination.");
    }

      // Activates agent itself when reaches destination host
    public void activate() throws BaseException {
        DefaultLogger.info(this, "Agent is activated in new host");
        if (SaferConst.USE_SAFER) continueKeyNegotiationWithNewHost();
    }
```

```
      // Add data provided by host to data segment
    public void addData(byte[] plain) throws BaseException {
        if (SaferConst.USE_SAFER) {
            addDataWithSAFER(plain);
        } else {
            addDataWithoutSAFER(plain);
        }
    }
      // Add ata provided by host to data segment (not using SAFER)
    private void addDataWithoutSAFER(byte[] plain) throws
BaseException {
        try {
            DefaultLogger.info(this, "Adding plain data into agent
without SAFER ...");
            m_agentSuitcase.addData(plain, getCurrentHostID());

        } catch (Throwable t) {
            if (t instanceof BaseException) throw (BaseException) t;
            t.printStackTrace();
            throw new BaseException("Exception in Agent addData");
        }
    }

      // Add data into agent's data segment (using SAFER)
    private void addDataWithSAFER(byte[] plain) throws BaseException
{
        try {
            DefaultLogger.info(this, "Adding plain data into agent
with SAFER ...");
            // format data: d + Id(host) + Id(agent) + ts
            Vector v = new Vector();
            v.addElement(plain);
            v.addElement(getCurrentHostID());
            v.addElement(getID());
            v.addElement(new Date());
            // encrypt data with DEK
            byte[] buf = ObjectUtil.objectToBytes(v);
            buf = CryptoUtils.pad(buf);
            IEncryptionService encryptor =
CryptoFactory.getEncryptionService();
            byte[] cipher =
encryptor.encrypt(getDEK(getCurrentHostID()), buf);

            // request host signature
            //HCP10Cient client = new
HCP10Client(getCurrentHost().getAddr(), getCurrentHost().getPort());
            //byte[] signature = client.requestHostSignature(getID(),
cipher);
            byte[] signature =
getCurrentHost().processDataSignatureRequest(getID(), cipher);
            // if attack scenario 1 enabled, randomly modify a byte
of the incoming data
            if (AttackUtils.isHostAttackEnabled(1,
getCurrentHost().getID())) {
                DefaultLogger.info(this, "Byte 0 was: " + cipher[0]);
                cipher[0] = (byte) (cipher[0] ^ 0x11);
                DefaultLogger.info(this, "Byte 0 is now: " +
cipher[0]);
            }

            // add data and signature into agent suitcase
            m_agentSuitcase.addData(cipher, signature,
getCurrentHostID());
            DefaultLogger.info(this, "Data added together with
signature successfully");
        } catch (Throwable t) {
            if (t instanceof BaseException) throw (BaseException) t;
            t.printStackTrace();
            throw new BaseException("Exception in Agent addData");
        }
    }

      // Send echo data back to agent butler
    public void echoData() throws BaseException {
        try {
            DefaultLogger.info(this, "Displaying data collected ...");
            for (int i = 0; i < m_agentSuitcase.getDataSize(); i++) {
                byte[] data = m_agentSuitcase.getDataAt(i);
                byte[] signature = m_agentSuitcase.getSignatureAt(i);
                String hostID = m_agentSuitcase.getHostIDAt(i);
                DefaultLogger.info(this, "\t" + (i+1) + ": From Host
" + hostID + "\t" + HexUtil.bytesToHex(data) + "\t" +
HexUtil.bytesToHex(signature));
            }
        } catch (Throwable t) {
            if (t instanceof BaseException) throw (BaseException) t;
            t.printStackTrace();
            throw new BaseException("Exception in Agent echo Data");
        }
    }

      // Verifies agent data integrity when the agent reaches a new
entity
    public void verifyDataIntegrity() throws BaseException {
        try {
            DefaultLogger.info(this, "Verifying data integrity ...");
            for (int i = 0; i < m_agentSuitcase.getDataSize(); i++) {
                byte[] data = m_agentSuitcase.getDataAt(i);
                byte[] signature = m_agentSuitcase.getSignatureAt(i);
                String hostID = m_agentSuitcase.getHostIDAt(i);
                DefaultLogger.info(this, "\t" + (i+1) + ": From Host
" + hostID + "\t" + HexUtil.bytesToHex(data) + "\t" +
HexUtil.bytesToHex(signature));
                // get public key of host
                RSAPublicKey pubKey = CertDB.getCert(hostID);
                // perform hash on data
                IMDService md = CryptoFactory.getMDService();
                byte[] hash = md.hash(data);
                byte[] hash2 = RSAUtils.publicDecrypt(signature,
pubKey);
                DefaultLogger.info(this, "Hash from storage   :\t" +
HexUtil.bytesToHex(hash));
                DefaultLogger.info(this, "Hash from calculated:\t" +
HexUtil.bytesToHex(hash2));
                if (hash.length != hash2.length) throw new
BaseException("Signature length mismatch " + hash.length + "\t" +
hash2.length);
                for (int j = 0; j < hash.length; j++) {
                    if (hash[j] != hash2[j]) throw new
BaseException("Signature value mismatch");
                }
            }
            DefaultLogger.info(this, "Signature verification
successful!");
        } catch (Throwable t) {
            if (t instanceof BaseException) throw (BaseException) t;
            t.printStackTrace();
            throw new BaseException("Exception in Agent
verifyDataIntegrity");
        }
    }
```

```
      // Get agent data at host i
    public byte[] getAgentData(int i) throws BaseException {
        return m_agentSuitcase.getDataAt(i);
    }

      // Get agent signature at host i
    public byte[] getAgentSignature(int i) throws BaseException {
        return m_agentSuitcase.getSignatureAt(i);
    }

    public void setAgentData(int i, byte[] data) throws BaseException
{
        m_agentSuitcase.setDataAt(i, data);
    }

    public void setAgentSignature(int i, byte[] signature) throws
BaseException {
        m_agentSuitcase.setSignatureAt(i, signature);
    }

    public int getAgentDataCount() throws BaseException {
        return m_agentSuitcase.getDataSize();
    }

    public void removeAgentRecordAt(int index) {
        m_agentSuitcase.removeDataAt(index);
        m_agentSuitcase.removeSignatureAt(index);
        m_agentSuitcase.removeHostAt(index);
    }

    public String getHostIDAt(int index) {
        return m_agentSuitcase.getHostIDAt(index);
    }
```

```
      // For attack simulation
    public void forgeItinerary(byte[] data, byte[] signature, String
hostID) throws BaseException {
        m_agentSuitcase.addData(data, signature, hostID);
    }
}
```

Source code of Butler.java:

```
package sg.edu.nus.safer.object;

import java.io.*;
import java.util.*;

import sg.edu.nus.base.logger.DefaultLogger;
import sg.edu.nus.base.crypto.CryptoFactory;
import sg.edu.nus.base.crypto.IKeyExchange;
import sg.edu.nus.base.crypto.IMDService;
import sg.edu.nus.base.crypto.IEncryptionService;
import sg.edu.nus.base.crypto.RSAUtils;
import sg.edu.nus.base.exception.BaseException;
import sg.edu.nus.base.util.PauseUtil;
import sg.edu.nus.base.util.HexUtil;
import sg.edu.nus.base.util.ObjectUtil;
import sg.edu.nus.safer.bat.BATProtocolHandler;
import sg.edu.nus.safer.bcp.BCPProtocolHandler;
import sg.edu.nus.safer.knp.KNPProtocolHandler;
import sg.edu.nus.safer.knp.KNPHelloMessage;
import sg.edu.nus.safer.knp.KNPReplyMessage;
import sg.edu.nus.safer.knp.KNP10Client;
import sg.edu.nus.safer.constant.SaferConst;
import sg.edu.nus.safer.cert.CertDB;
import sg.edu.nus.safer.util.ItineraryTracker;
import sg.edu.nus.safer.util.ItineraryObject;

import firefox.crypto.CryptoUtils;
import firefox.crypto.RSAPublicKey;
import object.Owner;
import cana.io.*;
import crypto.MasterCert;
import com.integrosys.base.techinfra.propertyfile.PropertyManager;



public class Butler extends Owner {

    private IKeyExchange m_keyExchange = null;
    private byte[] m_key_seed =
PropertyManager.getValue("butler.InitialKeySeed",
"supermanmagicgal").getBytes();
    private int m_skey_counter = 0;
    private Hashtable m_agents;
    private Hashtable m_agents_roaming;

      // Constructor
    public Butler(String ID, String description, MasterCert cert) {
        super(ID, description, cert);
        init();
    }

      // Initialize agent butler
    private int init() {
        DefaultLogger.info(this, "Butler " + getID() + "
initializing ...");
        try {
            // Cache DH parameter for potential KNP request
            initDH();
            m_agents = new Hashtable();
            m_agents_roaming = new Hashtable();
            return 1;
        } catch (Throwable t) {
            t.printStackTrace();
            return -1;
        }
    }

      // Initialize for DH key exchange
    public void initDH() throws BaseException {
        if (!SaferConst.USE_SAFER) return;
        DefaultLogger.info(this, "Initializing DH key exchange ...");
        m_keyExchange = CryptoFactory.getKeyExchange();
        m_keyExchange.initialize();
    }

    public int getPort() {
        return PropertyManager.getInt("butler.Port");
    }
      // Starts to listen for agent requests
    public void startService() {
        try {
            JMessageServer ms = new JMessageServer();
            ms.setPort(getPort());
            KNPProtocolHandler pHandler = new KNPProtocolHandler();
            pHandler.init();
            pHandler.setButler(this);
            BCPProtocolHandler pHandler2 = new BCPProtocolHandler();
            pHandler2.init();
            pHandler2.setButler(this);
            BATProtocolHandler pHandler3 = new BATProtocolHandler();
            pHandler3.init();
            pHandler3.setButler(this);
            ms.registerProtocolHandler(pHandler);
            ms.registerProtocolHandler(pHandler2);
            ms.registerProtocolHandler(pHandler3);
            ms.start();
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
      // Set key seed with agent
    public void setKeySeed(byte[] key_seed) {
        m_key_seed = key_seed;
        m_skey_counter = 0;
    }

    public byte[] getKeySeed() { return m_key_seed; }
    private int nextCounterValue() {
        DefaultLogger.info(this, "Getting next counter value,
increasing by one from " + m_skey_counter);
        return ++m_skey_counter;
    }
    public void decreaseCounterValue() {
        DefaultLogger.info(this, "Decreasing counter value, this
should be called only if it is necessary to rollback the counter
value in case of failure!");
        m_skey_counter--;
    }
      // Derive communications session key
    public byte[] getSessionKey(String host_id) throws BaseException
{
        IMDService md = CryptoFactory.getMDService();
        md.initialize();
        if (getKeySeed() != null) md.update(getKeySeed());
        if (host_id != null) md.update(host_id);
        md.update(new Integer(nextCounterValue()).toString());
        byte[] hash = md.digest();
        byte[] key = new byte[SaferConst.SKEY_LEN];
        System.arraycopy(hash, 0, key, 0, SaferConst.SKEY_LEN);
        return key;
    }
```

```
      // get data encryption key
    public byte[] getDEK(String host_id) throws BaseException {
        IMDService md = CryptoFactory.getMDService();
        md.initialize();
        if (getKeySeed() != null) md.update(getKeySeed());
        if (host_id != null) md.update(host_id);
        byte[] hash = md.digest();
        byte[] key = new byte[SaferConst.DEK_LEN];
        System.arraycopy(hash, 0, key, 0, SaferConst.DEK_LEN);
        return key;
    }
      // Process KNP hello message
    public KNPReplyMessage processHello(KNPHelloMessage hello) throws
BaseException {
        KNPReplyMessage reply = null;
        String agentID, targetHostID, targetHostAddr;
        int targetHostPort;
        if (SaferConst.USE_SAFER) {
            reply = processHelloWithSAFER(hello);
            agentID = (String) hello.getDataAt(2);
            targetHostID = (String) hello.getDataAt(5);
            targetHostAddr = (String) hello.getDataAt(3);
            targetHostPort = ((Integer)
hello.getDataAt(4)).intValue();
        } else {
            reply = processHelloWithoutSAFER(hello);
            agentID = (String) hello.getDataAt(1);
            targetHostID = (String) hello.getDataAt(0);
            targetHostAddr = (String) hello.getDataAt(2);
            targetHostPort = ((Integer)
hello.getDataAt(3)).intValue();
        }
        // update agent location
        ItineraryTracker.recordAgentMovement(agentID, targetHostID,
targetHostAddr, targetHostPort, getKeySeed());
        return reply;
    }

    private KNPReplyMessage processHelloWithSAFER(KNPHelloMessage
hello) throws BaseException {
        DefaultLogger.info(this, "Processing hello with SAFER ...");
        String currentHostID = (String) hello.getDataAt(0);
        String targetHostID = null;
        byte[] cipher = (byte[]) hello.getDataAt(1);
        String agentID = (String) hello.getDataAt(2);

        DefaultLogger.info(this, "Decrypting incoming cipher
message ...");
        IEncryptionService encryptor =
CryptoFactory.getEncryptionService();
        byte[]  plain  =  encryptor.decrypt(getSessionKey(currentHostID),
cipher);
        plain = CryptoUtils.unpad(plain);
        DefaultLogger.info(this, "Plain data size after decryption: "
+ plain.length);
        Object dh_public = null;
        try {
            Vector v = (Vector) ObjectUtil.bytesToObject(plain);
            targetHostID = (String) v.elementAt(0);
            dh_public = v.elementAt(1);
        } catch (Throwable t) {
            t.printStackTrace();
            throw new BaseException("Exception in Butler
processHelloWithSAFER ...");
        }
        DefaultLogger.info(this, "Performing DH key exchange
protocol ...");
        byte[] new_key_seed = m_keyExchange.negotiate(dh_public);
        dh_public = m_keyExchange.getPublicKey();
        PauseUtil.pause();
        if (!SaferConst.USE_NEW_KNP_FORMULA) {
            DefaultLogger.info(this, "Converting dh public param to
byte array for encryption ...");
            plain = ObjectUtil.objectToBytes(dh_public);
        } else {
            DefaultLogger.info(this, "Construct a vector with dh
public, ID host, ID agent, and timestamp");
            Vector v = new Vector();
            v.addElement(dh_public);
            v.addElement(targetHostID);
            v.addElement(agentID);
            v.addElement(new Date());
            plain = ObjectUtil.objectToBytes(v);
        }

        PauseUtil.pause();
        DefaultLogger.info(this, "Encrypting the dh param with the
target host " + targetHostID + "'s public key ...");
        DefaultLogger.info(this, "Plain data length is: " +
plain.length);
        RSAPublicKey host_pub_key = CertDB.getCert(targetHostID);
        cipher = RSAUtils.publicEncrypt(plain, host_pub_key);

        plain = cipher; // the encrypted dh_param will be further
encrypted by session key

        PauseUtil.pause();
        DefaultLogger.info(this, "Encrypting the result with new
session key ...");
        DefaultLogger.info(this, "Plain data size is: " +
plain.length);
        plain = CryptoUtils.pad(plain);
        cipher = encryptor.encrypt(getSessionKey(currentHostID),
plain);

        KNPReplyMessage reply = new KNPReplyMessage();
        reply.addData(cipher);
        setKeySeed(new_key_seed);
        DefaultLogger.info(this, "New Key Seed: " +
HexUtil.bytesToHex(new_key_seed));
        return reply;
    }
    private KNPReplyMessage processHelloWithoutSAFER(KNPHelloMessage
hello) throws BaseException {
        DefaultLogger.info(this, "Processing hello without
SAFER ...");
        String hostID = (String) hello.getDataAt(0);
        DefaultLogger.info(this, "Agent heading for " + hostID);
        KNPReplyMessage reply = new KNPReplyMessage();
        reply.addData("OK to roam");
        return reply;
    }

      // Process echo message from agent
    public void processEcho(String agentID, String hostID, byte[]
cipher) throws BaseException {
        DefaultLogger.info(this, "Processing echo message");
        byte[] plain = null;
        if (SaferConst.USE_SAFER) {
            IEncryptionService encryptor =
CryptoFactory.getEncryptionService();
```

```
            plain = encryptor.decrypt(getSessionKey(hostID), cipher);
            plain = CryptoUtils.unpad(plain);
        } else {
            plain = cipher;
        }
        DefaultLogger.info(this, "Agent " + agentID + " says: " + new
String(plain));
    }
      // Process echo request fro command line
    public void processEchoRequest(String agentID, String message)
throws BaseException {
        DefaultLogger.info(this, "Processing send echo message
request");
        byte[] cipher = null;
        if (SaferConst.USE_SAFER) {
            IEncryptionService encryptor =
CryptoFactory.getEncryptionService();
            byte[] plain = CryptoUtils.pad(message.getBytes());
            cipher =
encryptor.encrypt(getSessionKey(getAgentHostID(agentID)), plain);
        } else {
            cipher = message.getBytes();
        }
        KNP10Client client = new
KNP10Client(getAgentHostAddr(agentID), getAgentHostPort(agentID));
        client.sendBAEcho(agentID, getAgentHostID(agentID), cipher);
    }

    public String getAgentHostID(String agentID) throws BaseException
{
        //return "natalya";
        return ItineraryTracker.getAgentCurrentHostID(agentID);
    }
    protected String getAgentHostAddr(String agentID) throws
BaseException {
        //return "localhost";
        return ItineraryTracker.getAgentCurrentHostAddr(agentID);
    }
    protected int getAgentHostPort(String agentID) throws
BaseException {
        //return 2030;    // address of nat
        return ItineraryTracker.getAgentCurrentHostPort(agentID);
    }

    public Agent getAgent(String agentID) throws BaseException {
        try {
            Object agent = m_agents.get(agentID);
            return (Agent) agent;
        } catch (Throwable t) {
            t.printStackTrace();
            throw new BaseException("Exception from Butler getAgent");
        }
    }
      // Get all agents with the butler
    public Vector getAllAgents() {
        Vector v = new Vector();
        Enumeration enum = m_agents.keys();
        while (enum.hasMoreElements()) {
            String agentID = (String) enum.nextElement();
            v.addElement(agentID);
        }
        enum = m_agents_roaming.keys();
        while (enum.hasMoreElements()) {
            String agentID = (String) enum.nextElement();
            v.addElement(agentID);
        }
        return v;
    }
      // Process agent return request
    public void processAgentReturn(Agent agent) throws BaseException
{
        DefaultLogger.info(this, "Processing returning agent " +
agent.getID());
        DefaultLogger.info(this, "Performing itinerary
verification ...");
        // perform itinerary verification
        if (SaferConst.USE_SAFER) verifyAgentItinerary(agent);

        DefaultLogger.info(this, "Performing data verification ...");
        // perform integrity check
        if (SaferConst.USE_SAFER) agent.verifyDataIntegrity();
        m_agents.put(agent.getID(), agent);
        m_agents_roaming.remove(agent.getID());
        DefaultLogger.info(this, "Performing data decryption ...");
        PauseUtil.pause();
        if (SaferConst.USE_SAFER) decryptAgentData(agent);
    }
      // Decrypt data carried by returned agent
    private void decryptAgentData(Agent agent) throws BaseException {
        Vector v = ItineraryTracker.getAgentItinerary(agent.getID());
        for (int i=0; i<agent.getAgentDataCount(); i++) {
            byte[] cipher = agent.getAgentData(i);
            ItineraryObject itinerary = (ItineraryObject)
v.elementAt(i);
            byte[] keySeed = itinerary.getKeySeed();
            byte[] dek = getDEK(keySeed, itinerary.getHostID());
            IEncryptionService encryptor =
CryptoFactory.getEncryptionService();
            byte[] plain = encryptor.decrypt(dek, cipher);
            plain = CryptoUtils.unpad(plain);
            DefaultLogger.info(this, "Plain data at host (" + (i+1) +
") is: [" + (new String((byte[]) ((Vector)
ObjectUtil.bytesToObject(plain)).elementAt(0))) + "]");
        }
    }
      // reconstruct data encryption key at a particular host
    private byte[] getDEK(byte[] keySeed, String host_id) throws
BaseException {
        DefaultLogger.info(this, "Calculating DEK with host ID " +
host_id + " and key seed: " + new String(keySeed));
        IMDService md = CryptoFactory.getMDService();
        md.initialize();
        if (keySeed != null) md.update(keySeed);
        if (host_id != null) md.update(host_id);
        byte[] hash = md.digest();
        byte[] key = new byte[SaferConst.DEK_LEN];
        System.arraycopy(hash, 0, key, 0, SaferConst.DEK_LEN);
        return key;
    }

      // Add agent to roaming list
    public void addRoamingAgent(String agentID) {
        m_agents_roaming.put(agentID, agentID);
    }

      // Verifies agent integrity
    private void verifyAgentItinerary(Agent agent) throws
BaseException {
        Vector itinerary =
ItineraryTracker.getAgentItinerary(agent.getID());
        if (agent.getAgentDataCount() != itinerary.size()) throw new
BaseException("Agent Itinerary has been compromised - -1");
```

```
        for (int i=0; i < itinerary.size(); i++) {
            String host_id_in_agent = agent.getHostIDAt(i);
            String host_id_in_butler = ((ItineraryObject)
itinerary.elementAt(i)).getHostID();
            if (host_id_in_agent.equals(host_id_in_butler)) continue;
            else throw new BaseException("Agent Itinerary has been
compromised " + i);
        }
    }
}
```

Source code of Host.java:

```
package sg.edu.nus.safer.object;

import java.util.*;

import sg.edu.nus.base.exception.BaseException;
import sg.edu.nus.base.crypto.RSAUtils;
import sg.edu.nus.base.crypto.CryptoFactory;
import sg.edu.nus.base.crypto.IMDService;
import sg.edu.nus.base.util.ObjectUtil;
import sg.edu.nus.base.util.HexUtil;
import sg.edu.nus.base.util.PauseUtil;
import sg.edu.nus.base.logger.DefaultLogger;
import sg.edu.nus.safer.hcp.HCPProtocolHandler;
import sg.edu.nus.safer.knp.KNPProtocolHandler;
import sg.edu.nus.safer.bat.BATProtocolHandler;
import sg.edu.nus.safer.bat.BAT10Client;
import sg.edu.nus.safer.constant.SaferConst;
import sg.edu.nus.safer.attack.AttackUtils;
import sg.edu.nus.safer.demo.Jinx;
import firefox.crypto.RSAPrivateKey;
import crypto.MasterCert;
import cana.io.*;

public class Host extends object.Host {

    private Hashtable m_agents;

      // Constructor
    public Host(String ID, String description, MasterCert cert,
String addr, int port) {
        super(ID, description, cert, addr, port);
        m_agents = new Hashtable();
    }
      // Decrypt DH public key for agent
    public Object decryptDHPublicKey(byte[] cipher) throws
BaseException {
        try {
            RSAPrivateKey key =
((sg.edu.nus.base.crypto.rsa.RSAPrivateKey)
getMasterCert().getPrivateKey()).getRSAKey();
            DefaultLogger.info(this, "Cipher is " +
HexUtil.bytesToHex(cipher));
            DefaultLogger.info(this, "RSA Private Key is: " + key);
            byte[] plain = RSAUtils.privateDecrypt(cipher, key);
            Object dh_pub_key = ObjectUtil.bytesToObject(plain);
            return dh_pub_key;
        } catch (Throwable t) {
            t.printStackTrace();
            throw new BaseException("Exception in Host
decryptDHPublicKey");
        }
    }
      // Send agent to destination host
    public void sendAgent(Agent agent, String host, int port) throws
BaseException {
        DefaultLogger.info(this, "Agent " + agent.getID() + "
requests to be sent to host at " + host + ":" + port);
        try {
            // to invoke basic agent transport protocol
            BAT10Client client = new BAT10Client(host, port);
            client.sendAgent(agent);
            DefaultLogger.info(this, "Agent departed successfully");
            m_agents.remove(agent.getID());
        } catch (Throwable t) {
            t.printStackTrace();
            throw new BaseException("Exception in Host sendAgent");
        }
    }
      // Add agent into local agent list
    public void addAgent(Agent agent) throws BaseException {
        m_agents.put(agent.getID(), agent);
    }
      // Process incoming agent
    public void processIncomingAgent(Agent agent) throws
BaseException {
        DefaultLogger.info(this, "Processing incoming agent: " +
agent.getID());
        DefaultLogger.info(this, "Performing integrity check on
agent ...");
        PauseUtil.pause();
        // perform integrity check on datas carried by the agent
        if (SaferConst.USE_SAFER) agent.verifyDataIntegrity();

        DefaultLogger.info(this, "Integrity check completed,
accepting agent ...");
        PauseUtil.pause();
        m_agents.put(agent.getID(), agent);
        agent.setCurrentHost(this);
        agent.setCurrentHostID(getID());
        agent.activate();
        if (SaferConst.GUI_ENABLE) {
            // Code here is not elegant!!!

sg.edu.nus.safer.gui.host.HostMainFrame.getInstance().update();
        }
    }
```

```
      // Starting listening for incoming agent
    public void listenForIncomingAgent() throws BaseException {
        DefaultLogger.info(this, "Starting to listen for incoming
agent (blocking call) ...");
        try {
            JMessageServer ms = new JMessageServer();
            ms.setPort(getPort());
            BATProtocolHandler pHandler = new BATProtocolHandler();
            pHandler.init();
            pHandler.setHost(this);
            KNPProtocolHandler pHandler2 = new KNPProtocolHandler();
            pHandler2.init();
            pHandler2.setHost(this);
            HCPProtocolHandler pHandler3 = new HCPProtocolHandler();
            pHandler3.init();
            pHandler3.setHost(this);
            ms.registerProtocolHandler(pHandler);
            ms.registerProtocolHandler(pHandler2);
            ms.registerProtocolHandler(pHandler3);
            ms.start();
        } catch (Throwable t) {
            t.printStackTrace();
            throw new BaseException("Exception from Host
listenForIncomingAgent");
        }

    }

    // Received butler echo message for agent
    public void processEcho(String agentID, byte[] cipher) throws
BaseException {
        Agent agent = (Agent) m_agents.get(agentID);
        agent.processEcho(cipher);
    }

    // Received agent echo request for butler
    public void processEchoRequest(String agentID, String message)
throws BaseException {
        Agent agent = (Agent) m_agents.get(agentID);
        agent.sendEchoToButler(message);
    }
      // Process data signing request
    public byte[] processDataSignatureRequest(String agentID, byte[]
data) throws BaseException {
        // Perform hashing on data
        IMDService md = CryptoFactory.getMDService();
        byte[] hash = md.hash(data);
        // encrypt hash with private key
        RSAPrivateKey key =
((sg.edu.nus.base.crypto.rsa.RSAPrivateKey)
getMasterCert().getPrivateKey()).getRSAKey();
        byte[] signature = RSAUtils.privateEncrypt(hash, key);
        return signature;
    }
      // Process data deposit request
    public void processDepositDataRequest(String agentID, String data)
throws BaseException {
        Agent agent = (Agent) m_agents.get(agentID);
        agent.addData(data.getBytes());
    }
      // Process echo data request
    public void processEchoDataRequest(String agentID) throws
BaseException {
        Agent agent = (Agent) m_agents.get(agentID);
        agent.echoData();
    }
```

```
      // Process agent return request
    public void processAgentReturnRequest(String agentID) throws
BaseException {
        Agent agent = (Agent) m_agents.get(agentID);
        if (AttackUtils.isHostAttackEnabled(2, getID())) {
            // retrieve agent data collected from jinx at step 1
            byte[] data = agent.getAgentData(0);
            // modify agent data
            data[0] = (byte) (data[0] ^ 0x22);
            // forge signature with jinx's private key (somehow)
            byte[] signature = (new
Jinx()).processDataSignatureRequest(agent.getID(), data);
            // store information back to the agent
            agent.setAgentData(0, data);
            agent.setAgentSignature(0, signature);
        }
        if (AttackUtils.isHostAttackEnabled(3, getID())) {
            // remove data, signature, etc from AgentSuitcase
            agent.removeAgentRecordAt(0);
            // actually should reconstruct signature/hash tree as
well
        }
        if (AttackUtils.isHostAttackEnabled(5, getID())) {
            // add additional itinerary into the agent suitcase
            byte[] data = agent.getAgentData(0);
            byte[] signature = agent.getAgentSignature(0);
            String hostID = agent.getHostIDAt(0);
            agent.forgeItinerary(data, signature, hostID);
            // agentSuitcase.addData(data, signature, hostID);
        }
        BAT10Client client = new BAT10Client(agent.getButlerHost(),
agent.getButlerPort());
        client.sendAgent(agent);
        m_agents.remove(agentID);
    }

    public Vector getAllAgents() {
        Vector v = new Vector();
        Enumeration enum = m_agents.keys();
        while (enum.hasMoreElements()) {
            v.addElement(enum.nextElement());
        }
        return v;
    }

    public Agent getAgent(String agentID) { return (Agent)
m_agents.get(agentID); }
}
```

Appendix B

Major Source Code for SAT

Source code of MyAgent.java:

```
package examples;

import atp.*;

import stone.*;

import object.*;

import crypto.*;

import func.*;

import java.io.*;

import java.util.*;


/**

 * Demo Agent

 *

 * @version 1.0, 25 March 1999

 * @author Yang Yang (yyeung@usa.net)

 */

public class MyAgent extends IAgent {

      /**

       * Constructor

       *

       * @param ID Agent ID

       * @param description Agent description

       * @param cert Master certificate

       * @param ownerID Owner ID
```

```
	 * @param ownerEmail Owner's email address
	 * @see stone.IAgent
	 */
	public MyAgent(String ID, String description, MasterCert cert,
String ownerID, String ownerEmail) {
		super(ID, description, cert, ownerID, ownerEmail);
	}
	/**
	 * Overwrite agent invocation
	 */
	public void invoke() {
		super.invoke();
		System.out.println("\tThis is invocation implemented by
MyAgent");
		// Invoke open/secret function
		try {
			GenericFunction f = getFunction("OPEN");
			System.out.println("Function " + f.getFunctionID()
+ " loaded");
			f.invoke();
			f = getFunction("SECRET");
			System.out.println("Function " + f.getFunctionID()
+ " loaded");
			f.invoke();
		} catch (FunctionInvocationException e) {
			e.printStackTrace();
		}
	}
	/**
	 * Main routine (for testing)
	 */
	public static void main(String[] args) {
		String ID = "Agent 007";
```

```
            String description = "Bond, James Bond";
            String ownerID = "M";
            String ownerEmail = "yyeung@hotmail.com";
            MasterCert cert = null;
            try {
                  //ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(new
File("d:/master/transport/dat/camastercert.dat")));
                  ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(new
File("e:/master/transport/dat/camastercert.dat")));
                  cert = (MasterCert) ois.readObject();
            } catch (Throwable e) {
                  e.printStackTrace();
                  System.exit(-1);
            }
            MyAgent agent = new MyAgent(ID, description, cert,
ownerID, ownerEmail);
            GenericFunction f = new MyOpenFunction();
            f.setFunctionID("OPEN");
            agent.addFunction(f);
            f = new MySecretFunction();
            f.setFunctionID("SECRET");
            agent.addFunction(f);
            /*
            int i = agent.indirectSend("milan", 2002, "localhost",
2001);
            System.out.println("Status: " + i);
            */
            ATP10Client client = new ATP10Client("localhost", 2029);
            Vector v = client.requestEntryPermit("localhost", 2013,
agent);
            System.out.println("Request Entry Permit result: " +
v.elementAt(0) + "(" + v.elementAt(1) + ")");
            v = client.requestRoam("localhost", 2013);
            System.out.println("Request Roam result: " +
v.elementAt(0) + "(" + v.elementAt(1) + ")");
      }
}
```

Source code of MyHostA.java:

```
package examples;
import atp.*;
//import stone.*;
import metal.*;
import object.*;
import crypto.*;
import func.*;
import java.io.*;

/**
 * Demo Host A
 *
 * @version 1.0, 25 March 1999
 * @author Yang Yang (yyeung@usa.net)
 */
public class MyHostA extends IHost {
      /**
       * Constructor
       *
       * @param ID Agent ID
       * @param description Agent description
       * @param cert Master certificate
       * @param addr Host address
       * @param port Owner service port
       * @see stone.IAgent
       */
      public MyHostA(String ID, String description, MasterCert cert,
String addr, int port) {
            super(ID, description, cert, addr, port);
      }
      /**
```

```
        * Main routine (for testing)
        */
       public static void main(String[] args) {
             try {
                   String addr = "milan";
                   String ID = "Host - Singapore";
                   String description = "This is host Singapore saying
HI";
                   //ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(new
File("d:/master/transport/dat/host1mastercert.dat")));
                   ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(new
File("e:/master/transport/dat/host1mastercert.dat")));
                   MasterCert cert = (MasterCert) ois.readObject();
                   ois.close();
                   int port = 2029;
                  MyHostA a = new MyHostA(ID, description, cert, addr,
port);
                   a.startReceptionist();
             } catch (Throwable e) {
                   e.printStackTrace();
             }
       }
}
```

Source code of MyHostB.java:

```
package examples;
//import stone.*;
import metal.*;
import object.*;
import crypto.*;
import func.*;
import java.io.*;
import java.util.Vector;

/**
 * Demo Host B
 *
 * @version 1.0, 25 March 1999
 * @author Yang Yang (yyeung@usa.net)
 */
public class MyHostB extends IHost {
	/**
	 * Constructor
	 *
	 * @param ID Agent ID
	 * @param description Agent description
	 * @param cert Master certificate
	 * @param addr Host address
	 * @param port Owner service port
	 * @see stone.IAgent
	 */
	public MyHostB(String ID, String description, MasterCert cert,
String addr, int port) {
		super(ID, description, cert, addr, port);
	}
	/**
```

```
        * Main routine (for testing)
        */
      public static void main(String[] args) {
            try {
                  String addr = "milan";
                  String ID = "Host - London";
                  String description = "White Knight, White Knight,
com'in";
                  //ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(new
File("d:/master/transport/dat/host2mastercert.dat")));
                  ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(new
File("e:/master/transport/dat/host2mastercert.dat")));
                  MasterCert cert = (MasterCert) ois.readObject();
                  ois.close();
                  int port = 2013;
                 MyHostB a = new MyHostB(ID, description, cert, addr,
port);
                  a.startReceptionist();
            } catch (Throwable e) {
                  e.printStackTrace();
            }
      }
}
```

Appendix C

Detailed Message Format in SADIS Prototype

AB_HELLO: Hello message to initiate key seed negotiation. This message is sent by the agent to the agent butler. It contains the following:

String currentHostID:	ID of the current host (source host)
String targetHostID:	ID of the destination host
byte[] cipher:	Encrypted DH public parameter (for agent butler)
String agentID:	ID of the agent

BA_REPLY: Reply message in response to KNP_AB_HELLO message. It is sent by the agent butler to the agent. It contains the following:

byte[] cipher:	Encrypted DH public parameter (for agent)

AB_ECHO: Echo message from agent to agent butler in order to test encryption using communication session key. It contains the following:

String agentID:	ID of the agent
String hostID:	ID of the host where the agent is located
byte[] cipher:	Encrypted echo message

BA_ECHO: Echo message from the agent butler to agent in order to test encryption using communication session key. It contains the following:

String agentID:	ID of the agent
String hostID:	ID of the current host
byte[] cipher:	Encrypted echo message

AB_ECHO_REQ: Request message for agent to butler echo request. This is sent to the agent. It contains the following:

String agentID:	ID of the agent
String message:	Plain echo message

BA_ECHO_REQ: Request message for butler to agent echo request. This is sent to the butler. It contains the following:

String agentID:	ID of the agent
String message:	Plain echo message

DATA_SIGNATURE_REQ: Request from the agent sent to current host for PKI signature request. It contains the following:

String agentID:	ID of the requesting agent
byte[] data:	Data to be signed

DATA_SIGNATURE_REP: Reply from the current host in response to a data signature request. It contains the following:

byte[] signature:	Digital signature

DEPOSIT_DATA_REQ: Request message from the current host to deposit data to an agent. It contains the following:

String agentID:	Destination agent ID
byte[] data:	Data to be deposited

DEPOSIT_DATA_REP: Reply message in response to a data deposit request. It contains the status:

String status: Status of the request. Should be "OK" if the request is successful

ECHO_DATA_REQ: Request to display the data collected from an agent. This message is sent to the agent butler to decrypt and display data collected by a particular agent. It contains the following:

String agentID:	ID of the agent

ECHO_DATA_REP: Reply message to echo data request. This is sent by the agent butler to the requestor. It contains the status:

String status: Status of the request. Should be "OK" if the request is successful

AGENT_RETURN_REQ: Request message for an agent to return to butler. This is usually sent to a foreign host to request for the return of a particular agent. It contains the following:

String agentID:	ID of the agent

AGENT_RETURN_REP: Reply message in response to an agent return request. It contains the status:

String status: Status of the request. Should be “OK” if the request is successful