Securing Mobile Agents Through Evaluation of Encrypted Functions*

Hyungjick Lee†
Connectivity Lab., Digital Media R&D Center
Samsung Electronics Co., Ltd
416, Maetan-3Dong, Paldal-Gu,
Suwon City, Gyeonggi-Do, South Korea
hyungjick.lee@samsung.com

Jim Alves-Foss and Scott Harrison
Center for Secure and Dependable Software
Computer Science Department
University of Idaho
Moscow, ID 83843
{jimaf, harrison}@cs.uidaho.edu

Abstract

The mobile agent technology is a new paradigm of distributed computing that can replace the conventional client-server model. However, it has not become popular due to some problems such as security. The fact that computers have complete control over all the programs makes it very hard to protect mobile agents from untrusted hosts. In this paper we propose a security approach for mobile agents, which protects mobile agents from malicious hosts. Our new approach prevents privacy attacks and integrity attacks to mobile agents from malicious hosts.

Many people have proposed good security approaches, but most of them do not prevent both integrity and privacy attacks. We review a few security approaches for mobile agents, discuss their weaknesses and strengths, and propose a new approach that can fix many of their problems. One interesting approach is mobile cryptography proposed by Sander and Tschudin. It encrypts mobile agents and the encrypted mobile agents are executable without decryption. Implementing mobile cryptography requires an interesting types of cryptosystem called homomorphic encryption scheme, which allows direct computation on encrypted data, but none of such a homomorphic encryption scheme is known yet.

Our new security approach is an extension of mobile cryptography, and it removes many problems found in the original idea of mobile cryptography while preserving most of the benefits. Although the original idea of mobile cryptography allowed direct computations without decryptions on encrypted mobile agents, it did not provide any practical ways of implementation due to the fact that no homomorphic encryption schemes are found for their approach. Our approach provides a practical idea for implementing mobile cryptography by suggesting a hybrid method that mixes a function composition technique and a homomorphic encryption scheme that we have found. Like the original mobile cryptography, our approach will encrypt both code and data including state information in a way that enables direct computation on encrypted data without decryption. We believe that our approach is a viable and practical means to address security problems such as integrity and privacy attacks to mobile agents.

Keywords: *Mobile Agent System, Security, Encryption, Privacy, Homomorphism*

1 Introduction

An agent-based computer system is a distributed computing environment in which mobile autonomous processes called mobile agents operate on behalf of users. The autonomous agent concept has been proposed for a variety of applications on large, heterogeneous, distributed systems (e.g., the Internet) [5]. These applications include a specialized search of a large free-text database [3], middleware services such as an active mail system, electronic malls for shopping, and updated networking devices. Mobile agent systems are purported to have many advantages over traditional distributed computing environments. They require less network bandwidth, increase asynchrony among clients and servers, dynamically update server interfaces and introduce concurrency [4].

Mobile agents have existed for some time, but problems with security have limited their popularity. Mobile agents are composed of code, data, and state. Agents migrate from one host to another tak-

*This research was supported by research grants from DARPA under the contract number $DMA972-00-1-0001$ and Air Force Research Laboratory under the contract number $F30602-02-1-0178$.

†Corresponding Author: TEL +82 31 200 3354, FAX +82 31 200 3350

ing the code, data and state with them. The state information allows the agent to continue execution from the point where it was before it left in the previous host. For example, a mobile agent could be dispatched from the home site with the task of buying an airplane ticket for its owner. The agent would visit all the known hosts of airline companies, one after another, to search for the most reasonably priced ticket, and then purchase one for its owner. Each time the agent hops to the next host, it summarizes the current state, execution pointer on the current state, etc., so that it can start searching for reasonable tickets on the next host. The state of the agent will contain a set of possible tickets to be considered for purchase. When the agent has finished its search, it may return to the host where it found the cheapest or best ticket and purchase it.

While agents roam around the Internet, they are exposed to many threats and may also be a source of threat to others. Sander and Tschudin present two types of security problems that must be solved [13]. The first is host protection against hostile agents. The second is agent protection against hostile hosts. Many techniques have been developed for the first kind of problem, such as access control, password protections, and sand boxes, but the second problem appears to be difficult to solve. It is generally believed that the execution environment (host) has full control over executing programs; thus, protecting a mobile agent from malicious hosts is difficult to achieve unless some tamper-proof hardware is used. For example, Yee proposed an approach that uses a secure coprocessor that executes critical computations and stores critical information in secure registers [14]. In this paper, we propose a security approach to protect mobile agents from untrusted hosts.

In this paper, we focus on extending the mobile cryptography approach, proposed by Sander and Tschudin [11, 13, 12], in terms of privacy and integrity, and explore its usefulness and effectiveness in protecting mobile agents (We discuss the mobile cryptography in Section 2). To extend mobile cryptography, we will consider composite functions and additive-multiplicative homomorphism to encrypt mobile agents. As the contribution of this research, the encrypted mobile agent will be able to run on any host without decryption. The encrypted mobile agent will generate encrypted results, which will be decrypted by the agent owner. This will improve the overall security of the mobile agents.

In the remainder of this paper, we expand the idea of mobile cryptography. In Section 2 we provide an overview of some related works. In Section 3 we introduce the idea of homomorphic encryption scheme and function composition. In Section 4 our new approach for mobile agent security is discussed. In Section 5 we discuss the details of each component of our approach. In Section 6 we analyze our approach, and discuss the weaknesses and strengths of our approach. In Sections 7 and 8 we briefly give conclusion and some future works for our approach.

2 Related Work

Mobile agent protection is difficult due to a host's complete control over executing programs. While many approaches have been proposed to defend mobile agents from untrusted hosts, none adequately addresses every aspect of security. We survey four proposed approaches for the problem of mobile agent protection. The four approaches are chosen because each approach is very uniquely implemented and has strengths that other approaches do not have; we choose social control approach because it mimics our real society where badly behaved merchants are forced to go out of business. Partial Result Authentication Code approach is chosen due to the fact that it can protect results from mobile agents. Environmental key generation approaches are chosen because it uses cryptographic support for privacy attack prevention. Mobile cryptography approaches is chosen because it tries to scramble code and data together.

2.1 Social Control

Rasmusson and Jansson propose an approach that utilizes the reputation of each host to improve the security of a mobile agent system [8]. Their approach require a dedicated reputation agent and machine for building and maintaining the reputation list of all the participating hosts. The reputation of each host changes throughout its life span depending on its behavior. Although this approach does not require any other external security authority, all the participants join in punishing malicious hosts by ad-

justing their reputation evaluation as determined by agents.

This social control has advantages over other approaches. First, social control is a "soft" approach that accepts malicious hosts but identifies them and prevents them from causing more problems [8]. The second advantage is that it provides modest general coverage for mobile agents. Third, once security breach detection mechanisms are determined, it is relatively easy to implement.

However, social control also has many disadvantages. First, to punish a malicious host, we need to detect attacks; however, Rasmusson and Jansson provide no clear method/algorithm for detecting attacks on mobile agents [8]. It would be a complex and time-consuming process to develop detection algorithms for different kinds of attack models. Second, this approach requires a dedicated agent and machine that could be a potential point of failure. Once the security for any two of these is breached, a safe mobile agent environment cannot be guaranteed. Third, since the reputation record for all the hosts must be maintained, a storage problem is created as the growing number of hosts generates large numbers of reputation records. Fourth, reputation is based on host identifiers; if an intruder can masquerade as an acceptable host, there are no additional safeguards. Fifth, This approach is slow in changes and responses to new malicious hosts.

2.2 Partial Result Authentication Codes (PRAC)

Yee proposed an approach, Partial Result Authentication Codes (PRAC), which protects partial result with a Message Authentication Code (MAC) computed on partial results by using a secret key [15]. The agent originator (owner) and mobile agent are given a secret key for each host to be visited. The current secret key used to encrypt the partial result is destroyed before the agent migrates to the next host. Destroying secret keys before agent migration ensures that the previous partial results are secure and intact. Since the agent originator maintains the secret keys, the partial results can be verified on the originator's home site.

PRACs also provide a reasonable protection to mobile agent systems by focusing primarily on integrity issues in the mobile agent system. PRACs have several positive aspects. First, PRACs improve the integrity of partial results, because the secret key(s) used to create PRACs are destroyed before an agent's migration. Second, unless secret keys are compromised, the agent originator can pinpoint which malicious host attempted an attack through comparing the PRAC generated by the malicious host to the PRAC generated by the correct secret key stored in the agent originator. The third advantage of the PRAC approach is that it guarantees forward integrity, which states that even though the current host is malicious, all the previous partial results are safe, because the secret key for each previous host is destroyed before an agent's migration.

There are negative aspects to the PRAC approach also. First, it provides protection to partial results for mobile agents, but not to agent code and other aspects of the agent. Second, if the secret keys are compromised by malicious hosts, then those malicious hosts can read and modify any partial results. Third, although secret keys are destroyed before agent migration, it does not ensure that future results are secure, if a host is ever revisited by an agent.

2.3 Environmental Key Generation

The next approach, Environmental Key Generation proposed by Riordan and Schneier, generates the decryption key for an agent's encrypted code and data by searching through the execution environment [9]. The agent originator sends a cipher-text message (i.e., encrypted data and instructions) and a method for searching the environment for the data that is required to generate the decryption key. If the proper environmental data is found through the given data channel, then the key is generated to decrypt the encrypted mobile agent.

Environmental key generation has many strengths over other approaches. First, environmental key generation improves the integrity and privacy for agent code and data, which are both encrypted by the agent. Second, the decryption key is kept secure. The programmer can choose any kind of data channel that best suits the application such as a file system, Internet newsgroup, or e-mail. Even though the attacker may know which data channel the agent is searching, he or she must know which data portion of the data channel is required for the key generation.

The environmental key generation can protect the code and data from integrity and privacy attacks, but this approach also has weaknesses. First, the environmental key generation approach is vulnerable to group conspiracy attack. Second, data channel protection is another security issue. Third, although this approach can improve the integrity and the privacy for its code and data, it does not provide any protection for results. Fourth, once the code and data are decrypted, they can be attacked by a malicious host who can insert his or her own decrypting routine and data channel for new hosts.

2.4 Mobile Cryptography

Mobile cryptography, originally proposed by Sander and Tschudin [11, 13], is an approach that uses cryptographic techniques. The basic concept of this approach is as follows: Alice has a function $f()$ and wants to run this function $f()$ on Bob's computer with Bob's input x. Alice does not want Bob to know details of the function $f()$; and the result of the function $f()$ must not be revealed to Bob. To accomplish this, Alice carefully encrypts the function $f()$ into a new executable function $g()$ and transmits it to Bob. Since the function $f()$ is encrypted as $g()$, Bob does not know the details of the function $f()$. Bob runs $g()$ without decryption and generates results that are also encrypted. Bob sends the results back to Alice. Alice can decrypt the results to obtain the true results of $f()$. Sander and Tschudin argue that an additively, multiplicatively, and mixed-multiplicatively homomorphic encryption can implement the mobile cryptography which generates an encrypted mobile agent that will run without the decryption and generates the encrypted result.

Mobile cryptography offers many advantages over other approaches. First, it provides protection against privacy attacks and integrity attacks, while most of other approaches achieve protection for only one kind of attack. Since the agent is encrypted, the code, data and execution flow are protected from read attacks and modification attacks. Second, the partial results are also protected from read attacks and modification attacks since only the agent owner who encrypted the agent can decrypt the results. Third, there is no need to set up a secure channel to transmit agents because the agents are already encrypted. Fourth, the most powerful advantage of this approach is that the encrypted agent is executable and the partial results generated by the agent on any host are already encrypted. Only the agent owner can decrypt the results after the agent returns home.

Mobile cryptography provides the widest range of protection but also embodies some weaknesses. First, the most serious problem is that there are no known general encryption schemes for arbitrary functions. Sander and Tschudin proved that polynomial functions and rational functions can be encrypted in this way [13]. Second, there is also a small delay in agent creation and execution due to the encryption process. Third, if multiple hosts must participate in agent execution, then this approach is not useful because only the agent owner can encrypt and decrypt the results. This causes a problem in sharing information among multiple hosts involved in agent execution. Fourth, the encrypted agent may not be optimized. Unoptimized agents will show slow performance and consume a great amount of resources. Fifth, this approach provides a wide range of protection to mobile agents, but there are still some types of attacks that cannot be prevented, such as denial of service, agent hijacking, and random modification attacks.

3 Evaluating Encrypted Functions

Our approach is built on the bases of three-address code, homomorphic encryption scheme (HES), and function composition (FnC) technique. In this section, we describe three-address code, function composition (FnC) and homomorphic encryption scheme (HES) to prepare for our new approach.

3.1 Three-Address Code

Many computer languages use compilers to translate source code into executable target code. Compilers go through several phases; after the lexical, syntax and semantic analysis, some compilers, though not all, generate an explicit intermediate representation, before generating target code [1]. The three-address code is one of the forms of intermediate representations.

Three-address code is a sequence of the statements of the form $x := y \; op \; z$, where x, y, and z are names, constants or compiler-generated temporaries; *op*

stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like $x+y*z$ might be translated into a sequence:

$$t_1 := y * z$$
$$t_2 := x + t_1$$

where t_1 and t_2 are compiler generated temporary names [1]. In general, three-address code contains three addresses, where there are two addresses for the operands and one for the result.

3.2 Homomorphic Encryption Scheme (HES)

Rivest, Adleman and Dertouzos pointed out that the limitation of an encryption system is that an information system can only store and retrieve encrypted data for users. Further operations on data require decryption, and once the data is decrypted, it is not secure any more. Thus, the researchers proposed a new idea of cryptosystem that enables direct computation on encrypted data without decryption, which they called *privacy homomorphism* [10]. Later, Sander and Tschudin defined additive-multiplicative homomorphism, which is a kind of privacy homomorphism [13, 12]. Additive-Multiplicative homomorphism ensures that the computation result on two encrypted values is exactly the same as the encrypted result of the same computation on two unencrypted values. Sander and Tschudin's mobile cryptography uses HES for its implementation, but there are some drawbacks. First, no single cryptosystem is found to be additively, multiplicatively and mixed multiplicatively homomorphic. Second, only some limited classes of functions (polynomial and rational functions) are proved to be compatible with the HES [13, 12]. Here, we describe the properties of homomorphic encryption scheme that we need for securing mobile agents from the work of Sander and Tschudin [13, 12]:

> Let R and S be rings. We call an (encryption) function $E : R \rightarrow S$
>
> - *additively homomorphic* if there is an efficient algorithm **PLUS** to compute $E(x+y)$ from $E(x)$ and $E(y)$ that does not reveal x and y,
> - *multiplicatively homomorphic* if there is an efficient algorithm **MULT** to compute $E(xy)$ from $E(x)$ and $E(y)$ that does not reveal x and y,
> - *mixed-multiplicatively homomorphic* if there is an efficient algorithm **MIXED-MULT** to compute $E(xy)$ from $E(x)$ and y that does not reveal x.

The homomorphic encryption scheme that meets the three properties allow only two types of operators: addition and multiplication. One thing to note is that there is one-to-many relationship, which implies that a single plaintext message, x, can have multiple ciphertext messages of $E(x)$ (i.e., although $E_1(x) \neq E_2(x)$, $D(E_1(x)) = D(E_2(x))$ is true for a plaintext message x). Another point to note is that there should be only a few elements (only one element is desirable) that satisfies the last property (mixed-multiplicativity), otherwise the last property and the second property yield an anomaly, $y = E(y)$. Thus, in integers, only one integer (a multiplicative identity, $x = 1$) should satisfy the last property, $E(xy) = E(x)y$, to avoid the anomaly.

3.3 Function Composition(FnC)

Sander and Tschudin argue that evaluating encrypted functions (EEF) can be accomplished, not only by an additive and multiplicative homomorphism, but also by mathematical analogues such as composite functions [11]:

> Assume Alice wants to evaluate a linear map A at Bob's input x on Bob's computer. She does not want to reveal A to Bob, so she picks at random an invertible matrix S, computes $B := SA$ and sends B to Bob. Bob computes $y := Bx$ and sends y back to Alice. Alice computes $S^{-1}y$ and obtains the result Ax without having disclosed A to Bob.

We define $f(x)$ as a resultant composite function, if it is derived by taking the output of a function, $h(x)$, and using as the input to another function, $g(x)$. Mathematically, this is represented by $f(x) = g \circ h$ or $f(x) = g(h(x))$, where the function, $h(x)$, is a hidden (original) function. The function

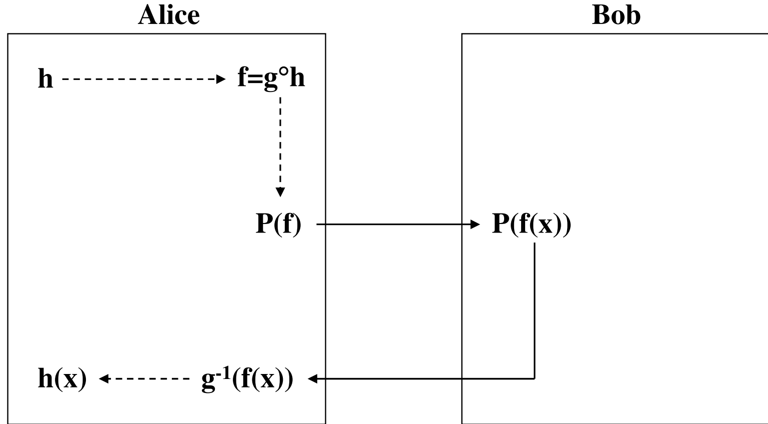

Figure 1: Composite Function

(agent) owner must choose an invertible function, $g(x)$, to create a composite function $f(x)$. The function, $f(x)$, is a different function (encrypted functions) from $h(x)$; thus, privacy and integrity requirements are preserved. The result of this composite function, $f(x)$, is also encrypted; malicious hosts do not know the result. The function (agent) owner retrieves the result from the encrypted result by using the inverse function of $g(x)$.

In Figure 1, Alice is the agent (function) owner and has a function, $h(x)$, that she wants to evaluate on Bob's computer with Bob's input x, but she does not want to reveal anything about her function. Alice chooses an invertible function, $g(x)$, creates a composite function, $f(x)$, and sends it to Bob. Bob does the computation with his input, x, and sends the result back to Alice. Bob cannot determine the function, $h(x)$, because what he sees is only the composite function $f(x)$. Only Alice can retrieve the real result of $h(x)$ from the result of $f(x)$ by plugging $f(x)$ into the inverse function of $g(x)$(i.e., $h(x) = g^{-1}(f(x))$).

4 New Approach

Ours is a hybrid approach which combines HES and FnC. A special program called *Mobile Agent Encryption (MAE)* will intercept the three-address code from compilers, and apply HES to encrypt the operands of three-address code and FnC to encrypt codes. In other words, *MAE* will encrypt the sensitive data, such as credit card number and personal information, stored in the operands of three-address code, and scramble the code of the mobile agent to confuse untrusted hosts. Our approach inherits most of the strengths of mobile cryptography; ours encrypt mobile agents, and the encrypted mobile agents are executable without decryption. The partial result is also protected by our approach. Furthermore, our approach removes some critical problems found in the original idea of mobile cryptography by Sander and Tschudin (this is discussed in analysis section). Implementing our approach requires assumptions and there are some limitation stemming from the assumption. Before, we present our approach, we state the assumptions and the goals of our approach.

4.1 Goals of Our Approach

The first goal of our approach is to enhance the privacy so that malicious hosts are not be able to read the contents of important data. The next goal is to enhance the integrity of the agent. Generally, hosts running programs have complete control over the programs; thus malicious hosts can read the mobile agent's code, analyze the flow of control, and modify the mobile agent. Because this will disrupt the normal execution of the mobile agent, the integrity of the result generated by the agent cannot be guaranteed. Another goal is encrypting the mobile agent carefully using an additive-multiplicative homomorphic encryption scheme so that the encrypted mobile agent is executable without decryption. Another goal is to protect the result generated from

the encrypted mobile agent. The results also suffer from the same security problems as the mobile agent. Without the protection of the result, malicious hosts can read and modify the result for their own benefits. The last goal is that no one except the agent owner must be able to decrypt the agent and result.

4.2 Assumptions for New Approach

Our approach offers broader protection to mobile agents than other approaches reviewed in Section 2. Implementing our approach requires some assumptions like the original mobile cryptography. The first assumption we need is that the HES is based on ring theory; thus we assume that the transformation (encryption/decryption) of elements from one set into the other set is additively and multiplicatively homomorphic. The second assumption is that we use only integers, due to the the fact our HES is based on ring theory. The third assumption is that only addition and multiplication are used in the agent code. Again, this is because we are using an additive-multiplicative homomorphic encryption scheme. The fourth assumption is that the control structures of the agent code will not be encrypted by the composite function technique, because the control structures such as `if`-statements have logical expressions with other types of operators such as logical operators, boolean operator and equality operators.

4.3 Encryption

Our new approach extends Sander and Tschudin's idea of mobile cryptography and overcomes the problems of their approach by proposing a practical way of implementing mobile cryptography. The new approach encrypts the data from the agent, then encrypts the three-address code representation of the agent by using the composite function technique. This implies that the mobile agent is doubly encrypted, by first encrypting the data and state information found in the three-address code with the additive-multiplicative homomorphic encryption scheme, and then by encrypting the three-address code (i.e., the code portion of the mobile agent) with the composite function technique. The overall workflow of the approach is a four-step process as in Figure 2:

- **Step 1** The operands of the three-address code are encrypted by using an additive-multiplicative homomorphic encryption scheme.
- **Step 2** The operand dependency problem (data encrypted by HES should not be encrypted again) caused by the additive-multiplicative homomorphic encryption scheme is removed.
- **Step 3** Three-address code statement is encrypted by using the function composition technique.
- **Step 4** The three-address code dependency problem caused by the function composition technique is resolved.

As shown in Figure 2, *MAE* grabs the plaintext three-address code from the compiler and analyzes the code for the encryption of sensitive data and state information of the mobile agent by using HES. After the first encryption, *MAE* encrypts the three-address code by using the function composition technique. While performing the first and second encryptions, *MAE* encounters the double encryption problem for the operands and the codes (statements). This will lead to the incorrect encryption of the mobile agents, thus *MAE* carefully looks for all the operands and statements for the double encryption, removing the double encryption, if any. The double encryption problem for each component (HES and FnC) is addressed in Section 5. *MAE* will generate the encrypted three-address code, which performs the same task as the plaintext three-address code, and makes it difficult for malicious hosts to read and modify the mobile agent code, data and state information.

4.4 Decryption

The process of decryption, as depicted in Figure 3, is exactly the reverse of the encryption process. The agent owner does not need to decrypt the whole mobile agent; instead only the encrypted result is decrypted. The result of the computation of any encrypted mobile agent is automatically encrypted, and malicious hosts cannot read and understand the encrypted result as depicted in Figure 3. The decryption done by the agent owner's *MAE* is a two-pass process, in which the first pass will use

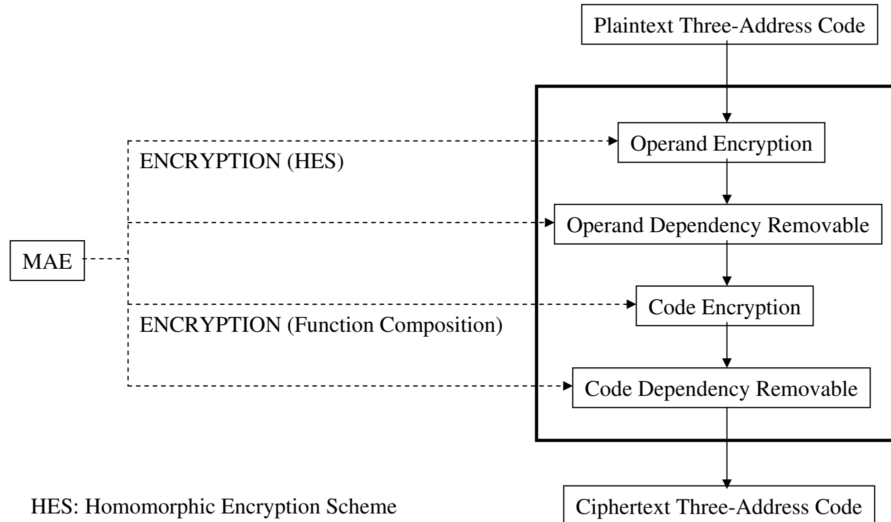

Figure 2: Encrypting Mobile Agent

the function composition technique to decrypt the result and the second pass will use the additive-multiplicative homomorphic encryption scheme to fully recover the actual result. The first pass will simply use the inverse of the function used to create the composite function during the encryption. The second pass uses the secret decryption keys of the additive-multiplicative homomorphic encryption scheme to obtain the actual result.

4.5 Overall Idea

Figure 4 depicts the overall process of encryption and decryption. During the encryption, HES is used to encrypt data, and a simple secret function, $g(x) = x^3 + 1$, is used for function composition. The encrypted mobile agent is released on the Internet, and returns with completed task (result) to the agent owner. In the decryption the inverse function, $g^{-1}(x) = \sqrt[3]{x+1}$, is used first, and HES is used to retrieve the actual result.

5 Components for Our Approach

In this section, we discuss the three organizing components of our approach, and provide the working details of each component.

5.1 HES

One of the biggest problems in mobile cryptography by Sander and Tschudin is that there is no known general encryption scheme (i.e., there is no homomorphic encryption scheme that meets the three properties described in Section 3). We suggest a modified cryptosystem that is additively, multiplicatively and mixed-multiplicatively homomorphic. This modified cryptosystem is based on the cryptosystem proposed by Ferrer and Joancomartí'. They argue that their cryptosystem is only additively and multiplicatively homomorphic [2]. We avoid the details of Ferrer and Joancomartí's cryptosystem and directly discuss our modified cryptosystem.

The modified cryptosystem is similar to the original cryptosystem, except that it is more or less a downsized version so that it can work on the ring. The modified version uses a large number, n, such that $n = p \times q$, where p and q are large prime numbers, and $p \neq q$. The set Q_p is defined as $\{a| \ (a \notin Z_p) \cap (a \geq p)\}$, where the plaintext set is Z_p, and the ciphertext set is Z_n. The types of operations defined are addition and multiplication on Z_p. The encryption and decryption algorithms are as follows:

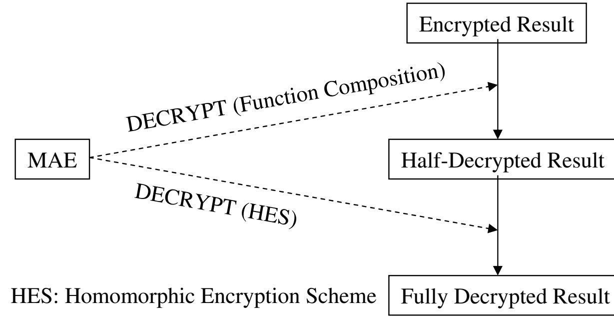

Figure 3: Decrypting Mobile Agent

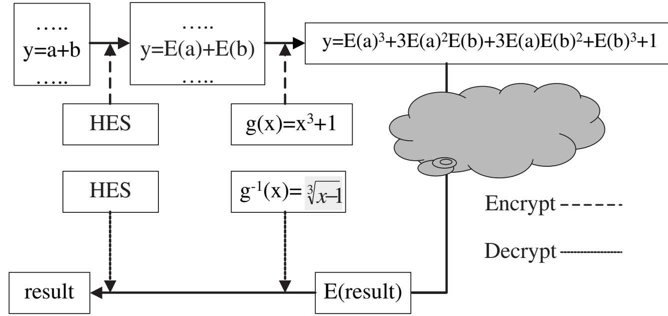

Figure 4: Overall Scheme

Encryption Given $x \in Z_p$, pick a random number a in Q_p such that $x = a \bmod p$. Compute $y = E_p(x) = a \bmod n$.

Decryption Given $y = E_p(x) \in Z_n$. Use the key p to recover $x = D_p(y) = y \bmod p$.

The modified cryptosystem is defined on ring and it is additively, multiplicatively, and mixed-multiplicatively homomorphic. It is easy to prove that the modified cryptosystem works correctly from Ferrer and Joancomartí's original proof [2].

Theorem 1 (Correctness) *For all $x \in Z_p$, $D_p(E_p(x)) = x$ holds true.*

Proof. Let $y = E_p(x)$ and a be the random number used to encrypt the message. From the definition, $y = a \bmod n$, we can deduce $a = nk + y$ for a constant k. Then the following holds true:

$$x = a \bmod p = (pqk + y) \bmod p = y \bmod p \quad (1)$$

□

The modified version is mixed-multiplicatively homomorphic in terms of the integer 1, as shown in the following:

Theorem 2 (Mixed-Multiplicativity) *For all s and t in Z_p where $s = 1$, $E(s)t = E(st)$ (i.e., $E(1)t = E(t)$) holds true.*

Proof. To prove the mixed-multiplicativity with $s = 1$, notice the encryption algorithm is $x = a \bmod p$, $y = a \bmod n$ for some plaintext $x \in Z_p$, and the decryption algorithm is $x = y \bmod p$ for some ciphertext y.
① $E(1)$:

$$1 = a_1 \bmod p, \quad a_1 = k_1 p + 1$$

$$y_1 = a_1 \bmod n, \quad a_1 = k_2 n + y_1$$

$$k_1 p + 1 = k_2 n + y_1, \quad y_1 = k_1 p - k_2 n + 1 = (k_1 - k_2 q)p + 1$$

Since $E(1) = y_1$, and $E(1)t = y_1 t$

$$y_1 t = (tk_1 - tk_2 q)p + t$$

implies

$$t = y_1 t \bmod p \tag{2}$$

② $E(t)$:

$$t = a_t \bmod p, \quad a_t = k_3 p + t$$

$$y_t = a_t \bmod n, \quad a_t = k_4 n + y_t$$

$$y_t = (k_3 - k_4 q)p + t$$

implies

$$t = y_t \bmod p \tag{3}$$

From equations 2 and 3,

$$y_1 t \bmod p = t = y_t \bmod p$$

This implies that $E(1)t = E(t)$ in terms of modulo p.
□

The properties of additivity and multiplicativity can be proven in a similar manner. The following example demonstrates the property of mixed-multiplicative homomorphism of the modified cryptosystem:

Example Assume $p = 101$, $q = 71$, and $n = pq = 7171$. Also assume the agent owner provides $E(1) = 203$. The malicious host wishes to encrypt the input, 8. Then, the malicious host multiplies $E(1)$ by 8, which yields the ciphertext, $E(8) = 1624$. To verify this, choose $A = 15966$, $15966 \bmod 7171 = 1624$ (Remember $A \in Q_n = \{A| \ (A \notin Z_n) \cap (A \geq n)\}$). Again $1624 \bmod 101 = 8$.
□

5.2 Double Encryption Problem

According to our design, *MAE* takes the plaintext three-address code to encrypt the operands of each statement. *MAE* will scan the three-address code file, and build the operand table to decide which operands to encrypt. This is done because if any of the operands are doubly encrypted, then the correctness of the computation from those operands is not guaranteed due to the fact $D(E(E(x))) \neq D(E(x))$. This fact implies that all the operands found in the intermediate file of the three-address code must be registered with *MAE* and the operands must be encrypted only once. The operands are variables, compiler-generated names and constants [1]. If any of the variables or constants are reused, then they must not be re-encrypted except when the program or mobile agent tries to assign a new value into the operands (i.e., variables).

Assume that we have the following three-address code:

$y := 10$
$a := y + 2$
$b := a + y$

The operands a, y, 2 and 10 are registered with *MAE* and encrypted. However, y in the second line, $a := y+2$, should not be encrypted because it already has the encrypted value, $E(10)$, from the first line ($y := 10$). Again, a in the third line, $b := a + y$, should not be re-encrypted, because a hold the encrypted value, $E(y) + E(2)$, from the second line.

5.2.1 Removing the Problem

Removing this problem requires a through scan of three-address code by MAE, and MAE builds an operand table internally, which contains all the operands including variables, temporaries and constants. Each table entry is associated with a boolean attribute, which has *true* if the corresponding entry (operand) is encrypted. Otherwise, it has *false*, which means that the corresponding entry is not encrypted (*false* is the default). As MAE encrypts each operand using HES, it assigns *true* to the corresponding entry. This continues until MAE finishes encrypting the last operand in the three-address code. While encrypting operands, MAE checks the operand table to see if the current operand (operand to encrypt) is already encrypted. If it is encrypted, then skips the encryption.

The following is an algorithm that removes the double encryption problem used by MAE. The assumption is that the next input (operand to encrypt) is fed into MAE from the right-most operand into the left-most operand (i.e., from $y = a+b$, read b, a, and y in the order). This algorithm stops when there is no more statements to process in the three-address code.

5.3 FnC

Once *MAE* is done with encrypting sensitive data using our additive, multiplicative, and mixed-multiplicative HES, it prepares for the next encryp-

Alg. 1 Algorithm to Remove Doubly Encrypted Operands

```
1: while Next_Operand do
2:   if constant then
3:     encrypt and mark with true
4:   else if encrypted then
5:     if reassignment then
6:       re-encrypt and mark with true
7:     end if
8:   else if right_side then
9:     encrypt and mark with true
10:  else if left_side then
11:    mark with true
12:  end if
13: end while
```

tion by the function composition. *MAE* decides a secret function, $g(x)$, which is $x^3 + 1$ in Figure 5, and uses it for each line of three-address code to encrypt. Figure 5 shows how to encrypt each three-address code line. It encrypted $y = a + b$ into a different form of code, $y = a^3 + 3a^2b + 3ab^2 + b^3 + 1$. *MAE* will replace each original three-address code with newly encrypted three-address code generated by random ordering of additions for each term and multiplications among the terms. This encryption process will repeat for the remaining three-address code lines. The decryption requires our *MAE* to use the inverse function of $g(x)$. Figure 5 shows the decryption on three-address code, but generally *MAE* performs the decryption on some results rather than the code.

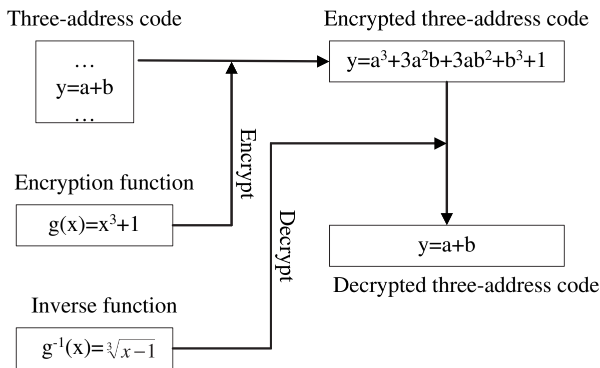

Figure 5: Encryption and Decryption Using the Function Composition

5.3.1 Double Encryption Problem Revisited

The function composition technique can encrypt each three-address code line and hide the details of the flow of control from malicious hosts and people, but encryption by function composition causes another double encryption problem caused by double encryption on statements (codes). The nature of the double encryption problem from function composition is similar to that of the HES, however it must be resolved differently. A simple example of the double encryption problem is as follows:

Example Assume a sequence of three-address codes $y = a + b$, and $z = y * c$. Also assume $g(x) = x^3 + 1$, $a = 1$, $b = 2$ and $c = 3$ for simplicity. The agent owner is expecting $z = 9$ from $(a+b)*c = (1+2)*3$, and he/she encrypts the code with the function composition into $y = a^3 + 3a^2b + 3ab^2 + b^3 + 1$ and $z = y^3c^3 + 1$ respectively (These are not in three-address code format, but *MAE* will convert them into corresponding three-address code, which does the same computation). Using $a = 1$, $b = 2$, and $c = 3$ onto the encrypted three-address code yields $z = 328510$, which is decrypted to $z = 69$ by the agent owner using the inverse function of $g(x)$. This is because y is encrypted twice in the first line ($y = a+b$) and again in the second line ($z = y * c$).

5.4 Our Solution

The solution to double encryption can be found by observing the behavior of compilers such as *LANCE* tool. LANCE is a tool, developed to retarget programs into different platforms, that generates three-address code from source codes written in C Language [6, 7]. The analysis on the behavior of LANCE tool is as follows:

Observation 1 LANCE uses original variable names with unique sequential numbers in three-address codes as shown in Figure 6. The numbers attached to the original variables help distinguish two variables with the same name.

Observation 2 LANCE tool generates temporaries such as $t1, t2, \cdots, tn$ in a way that avoids the reuse of each temporary as shown in the following example code:

$t_4 = jaeyong_100 + michelle_101$
$t_5 = t_4 + tim_102$
$yong_103 = t_5$

LANCE generated the three-address code in Figure 6 for $yong = jaeyong + michelle + tim$. As is clear, LANCE segmented the original expression with three terms so that each line in the three-address code can have up to two terms. Each intermediate result is stored into the temporaries t_4, and t_5.

5.4.1 Dependency-Tree Graph and Encryption

We have developed a scheme to avoid the double encryption problem caused by the function composition. The idea behind this scheme is to build a dependency-tree graph for three-address codes (i.e., a dependency-tree graph for each original variable found on the left side of any three-address code statements). The dependency-tree graph is a binary tree that has the first and second operands in its branches, an operator in the parent node, and an original variable in the root node. For each original variable, *MAE* builds a dependency-tree graph that is located on the left side of any three-address codes (i.e., y in $y = a + b$). Since there are seven original variables on the left side of the expression in Figure 6, *MAE* will generate seven dependency-tree graphs. Figure 7 shows a dependency-tree graph for the original variable, $yong_103$.

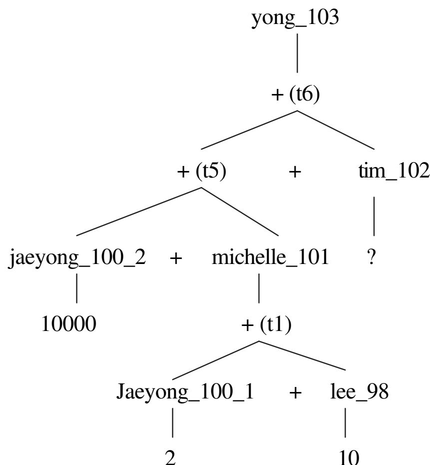

Figure 7: Dependency Graph

From Figure 7, it is clear that the leaf node always has a value corresponding to a source code variable. Sometimes the leaf nodes will have non-determined values (represented by a question mark), which means that the values are provided by untrusted hosts during run-time. From the dependency-tree graph in Figure 7, an expression $(jaeyong_100_1 + jaeyong_100_2 + lee_98 + tim_102)$ is constructed. In the dependency-tree graph of Figure 7, one variable, $jaeyong_100$ appears twice in two different nodes. One has an older value, 2, while the other has a newer value, 10000. To represent this situation, *MAE* modifies the variable names (there are two $jaeyong_100$'s) by adding a sequence number at the end, which requires us to insert extra statements, $jaeyong_100_1 = 2$ and $jaeyong_100_2 = 10000$ shortly before their use. This can be done without much work through *MAE*, while building a dependency-tree graph. To encrypt, *MAE* will encrypt $(jaeyong_100_1 + jaeyong_100_2 + lee_98 + tim_102)$ to obtain an expanded expression of $(jaeyong_100_1 + jaeyong_100_2 + lee_98 + tim_102)^3 + 1$. *MAE* will remove the three-address codes that are used to build the current dependency-tree graph for the variable, $yong_103$. Newly encrypted three-address codes from the expanded expression will be inserted into the location of the code $yong_103 = t6$. To confuse untrusted hosts further, the order of multiplications in each term will be randomly reorganized in a way that avoids disruption of the computation. The order of additions between two terms will also be randomly reorganized. Since there are seven original variables in the original three-address code, it will create seven blocks of codes (the number of blocks is not the number of original variables in the three-address code, but rather it is the number of the original variables that exist on the left side of three-address code lines). While replacing the codes, our new encrypted intermediate code file will have seven blocks of new statements (encrypted), in addition to the statements that assign static values, and dynamic values into original variables.

5.5 Case Study

Although encryption on three-address code follow the general encryption process, there are some cases that require some consideration.

```
1:
2:  /* 1 "LANCEV2" */
3:  /* $
4:       IR file generated by LANCE V2.0
(compile) on
5:       Monday, May 13, 2002 at 12:54:42
6:  $ */
7:
8:
9:  void main();
10:
11: static char *lance_static_t3 = "%d";
12:
13: void main()
14: {
15:  int lee_98;
16:  int sung_99;
17:  int jaeyong_100;
18:  int michelle_101;
19:  int tim_102;
20:  int yong_103;
21:  int t1;
22:  int t2;
23:  int t4;
24:  int t5;
25:  int t6;
26:
27:
28:  /* 6 "test.c" */
29:  /* $ int lee=10, sung=1, jaeyong=2,
michelle, tim, ...g$ */
30:
31:   lee_98 = 10;
32:   sung_99 = 1;
33:   jaeyong_100 = 2;
34:
```

```
35:  /* 9 "test.c" */
36:  /* $ michelle=jaeyong + lee;$ */
37:
38:   t1 = jaeyong_100 + lee_98;
39:   michelle_101 = t1;
40:
41:  /* 11 "test.c" */
42:  /* $ yong=lee+sung;$ */
43:
44:   t2 = lee_98 + sung_99;
45:   yong_103 = t2;
46:
47:  /* 14 "test.c" */
48:  /* $ jaeyong=10000;$ */
49:
50:   jaeyong_100 = 10000;
51:
52:  /* 17 "test.c" */
53:  /* $ scanf("%d", tim);$ */
54:
55:   t4 = scanf(lance_static_t3,tim_102);
56:
57:  /* 19 "test.c" */
58:  /* $ yong=jaeyong+michelle+tim;$ */
59:
60:   t5 = jaeyong_100 + michelle_101;
61:   t6 = t5 + tim_102;
62:   yong_103 = t6;
63:
64:  /* 0 "???" */
65:  /* $ ???$ */
66:
67:   return;
68: }
```

Figure 6: Example of Three-Address Code Generated by LANCE

First of all, line 39 of Figure 6 creates another dependency-tree graph for *michelle*_101, which is included(as a sub-tree) in the dependency tree graph of Figure 7. Although it is the same dependency-tree graph, it must be constructed again for line 39, and must replace the original set of three-address codes. This is because *MAE* does not know which variable is important. The original variables after full encryption (HES + FnC) must be available at any point through the life-time of mobile agent, because we do not know when we need to use a fully encrypted variable. For instance, we can print a variable now, and feed it into some other computations as an input later. Thus, *MAE* tries to build a dependency tree graph for every original variable, and replace original codes with encrypted three-address codes.

MAE builds dependency-tree graphs for original variables to encrypt the three-address code that is encrypted by HES. We should note that the encryption by function composition (FnC) is done independently for each dependency-tree graph. We should also note that a single variable can feed in the very same value into several independent dependency-tree graphs. This causes a problem in which the variable modified in the first dependency-tree graph feeds in wrong values into the subsequent dependency-tree graphs. If an original variable is used as an input in a later block (different dependency-tree graph), its original value (half-encrypted by HES) must be saved for the later blocks. Otherwise, it will feed in a wrong input and generate a wrong result. To prevent this problem, MAE inserts an extra statement to assign the value of an original variable (half-encrypted by HES) into a temporary and use this temporary in the later blocks (different dependency-tree graph). This insertion of extra code can be done before building dependency-tree graphs.

Another case to consider is when a new value is re-assigned to an original variable. In lines 33 and

50, new values 2 and 10000 are assigned into the variable, *jaeyong_100*, which will lead to two different dependency tree graphs, which must be built, and which must replace each original three-address code. Although this may appear to be redundant, it is not, due to the fact that the variable *jaeyong_100* with the value 2 can be used before the value, 10000, is assigned into the variable (for example, a simple print function `printf("%d", jaeyong_100)` may be called before assigning 10000 into the variable).

Another case that must draw our attention is that an input from malicious hosts should be dealt with very carefully. The input is usually accompanied by `scanf` command in C language. A variable is passed to `scanf` as a parameter, and the user input is stored in the variable. To encrypt the user input in the variable, our *MAE* should add some extra codes. Assuming that the input variable is a, our *MAE* adds $t1 = a * E(1)$, which encrypts a and stores the encrypted input in a temporary $t1$ by the definition of mixed-multiplicativity. The problem is that if we encrypt this line $t1 = a * E(1)$ by our secret function, $g(x) = x^3 + 1$, then it is possible that malicious hosts can discover our secret function by looking carefully at the three-address codes. To prevent this problem, we define a function named `split(first_half, second_half)`, where the parameter `first_half` is user input. The `split` function will randomly split the source input in `first_half` into two halves, and return them in `first_half` and `second_half`. Our *MAE* will multiply $E(1)$ to each half to encrypt each half, and add the two encrypted halves. Later, a dependency-tree graph is constructed for this, and will be encrypted through our function composition technique. Since there are two operands, `first_half` and `second_half`, the expanded expression from the dependency-tree graph becomes a bit complicated and can gain some time to confuse malicious hosts.

The next case involves simple assignment statements with a value on the right side of three-address codes as shown in lines 31, 32, and 33. Although there are no computations involved, the encrypting function, $g(x)$, will encrypt these simple assignments, and replace them with encrypted new three-address codes. Each line (31, 32, and 33) will build its own dependency tree graph. However, they must be treated in the same way as the previous case (use `split` function to confuse untrusted hosts). The reason for using `split` function is that malicious hosts may discover the secret function used in our function composition by carefully looking at the statements around the assignment. This kind of guessing attack is easier, if there is only one term to encrypt by secret function, but two or more terms to encrypt will generate longer and more confusing statements.

The dependency-tree graph can remove the double encryption problem in the three-address code for our function composition. However, there is one drawback: it is very hard to build a dependency-tree graph for an original variable, if there is a function call to a user-defined function, which should be another assumption of our approach; however, simple function calls for input and output, such as `printf` and `scanf`, do not cause problems.

6 Analysis

This chapter provides in-depth analysis of various aspects of the components, overall approach, and possible applications with restrictions. In the component analysis, we analyze each component, including homomorphic encryption, function composition, *MAE*, and the code growth problem. In addition, we will provide an overall approach analysis of its weaknesses and strengths.

6.1 Component Analysis - HES

6.1.1 Ring Theory

The homomorphic encryption scheme (HES) used in our approach is additively, multiplicatively, and mixed-multiplicatively homomorphic, and is based on ring theory. By nature, any number sets (integers) belonging to the ring allow only addition and multiplication. Another limitation from ring theory is that the control structures, such as `if`-statement, cannot be encrypted using HES, since ring theory allows only addition and multiplication. The next limitation is that there are only a few numbering sets (i.e., integers and integers with modular operation) that belong to the ring, due to the fact that there are only two operators defined (addition and multiplication), which limits the data type to integers.

6.1.2 Encryption

Any sensitive data, such as credit card numbers or some query information, are encrypted by the HES that allow the direct computation on encrypted data, if the computations involve only addition and multiplication. Any inputs to the encrypted mobile agent should be encrypted properly so that the encrypted mobile agent can perform computations on the encrypted inputs.

One of the advantages of our modified cryptosystem is that there is no need to release the encryption algorithm and the encryption keys. Another advantage is that the encryption of the input data from malicious hosts is automatically performed by mobile agents. A mobile agent includes some additional statements, which encrypt user input through a simple multiplication by $E(1)$. This actually requires only four extra lines of three-address code to encrypt user inputs (i.e., `split` function call, encrypting the first half and second half by E(1), and adding two encrypted halves). Without using `split`, it adds only one or two extra statements, but it may be possible that malicious hosts can grab our secret functions by analyzing the surrounding statements around the input statement (encrypting $y = a + b$ will give more confusion to malicious hosts than encrypting $y = a$). This encryption is very simple, and incurs very little overhead, thus yielding faster encryption performance when compared to normal encryption schemes that require the encryption algorithm and keys. The last advantage is that since this type of encryption has little overhead in mobile agents, these encrypted mobile agents can be used in an environment where real-time response is required.

6.2 Component Analysis - MAE

MAE grabs three-address codes, and encrypts twice, first by our HES, and second by our function composition technique. While encrypting, *MAE* prevents a double encryption problem for the HES used in our approach, by simply building an internal table for all the variables and temporaries in the three-address code. This requires *MAE* to look through each variable and temporary so that *MAE* can decide which one to encrypt. This process may cause some delays in encrypting operands, but this delay can be minimized by observing the behavior of compilers such as LANCE tool. The LANCE tool generates temporaries because of the limitation on the number of addresses in a three-address codes; there are at most two operands (i.e., addresses) on the right side. If a single code line in C language exceeds more than two addresses (operands), then temporaries are generated to hold intermediate results. From this observation, it may be possible to increase the speed of operand encryption by considering only the original variables on the left side of the three-address code and the constants; the temporaries will hold the encrypted result of addition and multiplication between two encrypted operands; thus, there is no need to encrypt them.

6.3 Component Analysis - FnC

In this sub-section, we analyze the function composition technique used in our approach.

6.3.1 Code Growth

Our approach intercepts and modifies the intermediate code in three-address form twice, and this process is expected to increase the size of code. The first modification will not increase the code size significantly, because *MAE* will replace only the unencrypted operands with the encrypted ones. However, we encrypt certain types of operands of three-address codes in a different way; these operands are dynamic inputs from malicious hosts and constant assignments, which will add some extra statements to the original three-address codes.

Unlike the first encryption, the second encryption performed by function composition adds many lines to the original three-address codes. Encryption by function composition creates several blocks of encrypted codes, depending on the number of original variables on the left side of three-address codes. Dependency-tree graphs created by *MAE* create several blocks of encrypted codes; each block is not directly related. If an original variable in a previous block were to be used in some other blocks, the original variable must be saved in a temporary variable after the value is encrypted by HES; the later blocks can use this new temporary to access the original value, which yields an extra code added into three-address codes for this purpose. From each dependency-tree graph created for each original variable comes an expanded expression, which requires an encryption through a secret function, $g(x)$

($f(x) = g(h(x))$, where $h(x)$ is an original code, and $f(x)$ is an encrypted code.

For the analysis of code growth, we will pick a worst case from the viewpoint of constant folding, which is one of many methods of optimizing intermediate codes. The idea of constant folding is simple: compilers can pre-evaluate an expression while compiling, if the expression has terms that are composed of constants [1]. This constant folding technique may pre-evaluate the whole expression, or a part of an expression, depending on the number of constants in three-address codes. Although the constant folding can reduce the size of the code by pre-evaluating three-address codes with constants, it is also true that this may not work well if there are no constants at all. Thus, from this fact we pick a worst case where this constant folding will not work.

The secret function, $g(x)$, plays a key role in increasing the size of the code in the second encryption phase. Although we have many choices for $g(x)$, a complicated secret function will greatly increase the size of code. If our secret function, $g(x)$, has a very high degree and many terms, it will greatly increase the size of the code, and give more confusion to malicious hosts. This will allow more time for secure computing for the encrypted mobile agent. On the other hand, if $g(x)$ is very simple, then it adds a lower number of codes to confuse malicious hosts.

We can obtain the upper bound for the size of encrypted three-address code in the worst case. Assume that there are k blocks (dependency-tree graphs), m terms for expanded expressions from dependency-tree graphs, and the secret function is $g(x) = \sum_{i=0}^{n} a_i x^i$. Homomorphic encryption function does not increase the code, because it encrypts operands and replaces them in the tree-address code. Thus, we do not need to consider our HES, but function composition. The expected number of encrypted statements for one expanded expression will be $\sum_{i=0}^{n} a_i m^i$ in the worst case. Since there are k dependency-tree graphs, the upper bound in the worst case is $k \sum_{i=0}^{n} a_i m^i$. This function is an upper bound for code growth, which does not include the variable declarations.

Even though the function composition greatly increase the code, there is still a chance to reduce the code size by grouping the same terms. For example, from $(a+b)^3 = a^3+a^2b+a^2b+a^2b+ab^2+ab^2+ab^2+b^3$, we can group the same terms together to obtain $a^3+3a^2b+3ab^2+b^3$. After this reduction, *MAE* can randomly perform multiplications for each term, and additions for the given expression. There are many other ways to reduce the size of code such as constant folding, which we will discuss in the next section.

6.3.2 Optimization

Usually, compilers have an extra step after an intermediate form of code is generated; one of extra steps is for the optimization on intermediate codes. There are many ways of optimizations: constant folding, common subexpression removal, copy propagation, variable renaming, and dead code elimination [1]. The main goal of function composition used in our approach is scrambling code so that mobile agents can earn time for secure computing. Although our function composition technique increases the size of code, it is possible that some of the code growth problems can be eliminated by using compiler optimization techniques.

We have tried to find a mathematical model that shows the difference in the code size between unoptimized codes and optimized codes; however, due to the random behavior of our encryption scheme (function composition), it was very difficult to find such a mathematical model. As our example indicates, the randomness of function composition (i.e., the random ordering of additions and multiplications) can limit the possible optimization techniques, and the range of optimizations. Even though all the optimization techniques may fail to reduce the code size in the worst case, there still is one optimization technique that can always reduce the size of code. The process of simplifying terms can reduce the code size and this will always work. MAE performs this term simplification while encrypting the three-address code with the function composition. However, this does not help to find a mathematical model that best expresses the size of codes between optimized and unoptimized codes due to the fact that the number of terms in expanded expression, m is variable in $k \sum_{i=0}^{n} a_i m^i$ (the size of encrypted code), where k is for the number of dependency-tree graph, and m for the number of terms in expanded expressions.

6.4 Security Analysis

In this sub-section we provide security analysis for our approach; we discuss the security of each component.

6.4.1 HES

Sander and Tschudin paved the way for mobile cryptography with one big limitation that there is no HES available for their mobile cryptography today. However, it turned out that it is not true, and there is a simple homomorphic encryption scheme. Ferrer and Joancomartí proposed an additive and multiplicative homomorphic encryption scheme. Our modified cryptosystem from the original one (Ferrer and Joancomartí cryptosystem) may not be perfectly secure, but there exists at least one additive, multiplicative and mixed-multiplicative homomorphic encryption scheme, and we can use this to extend, find or create a more secure one. Ferrer and Joancomartí discussed about the security of their cryptosystem. Since our modified version is a downgraded version and still shares many properties with the original version, we can use their arguments on the security analysis for our modified cryptosystem.

Ciphertext-Only Attack The cryptanalyst does not need p to find a number $A \in Q_n$ corresponding to a ciphertext $y \in Z_n$. However, p is needed to compute $a \bmod p = x$. But, if the cryptanalyst sees only ciphertext, then finding the secret p from the public n is as difficult as factoring n [2].

Known-Plaintext Attack If the cryptanalyst knows a plaintext-ciphertext pair (x, y), then the cryptanalyst can generate a set of the n numbers of $A_i \in Q_n$ for $i = 1, \cdots, n$ such that $A_i = y \bmod n$. Then, the cryptanalyst knows that $A_i = x \bmod p$ so that $p \mid (A_i - x)$ for all $i = 1, \cdots, n$.

The known-plaintext attack analysis implies that the modified cryptosystem as well as the original one are not perfectly secure. It is possible that our modified cryptosystem can be broken with a time-consuming effort; however, we can prevent this attack by modifying the assumption. If malicious hosts know n, then they can perform the known-plaintext attack as described above. However, if we keep n and p secret, this known-plaintext attack can be prevented. Since, the encryption code is already included in the encrypted mobile agents, the agent owner does not need to release the secret keys including n, and the encryption algorithm.

6.4.2 Function Composition (FnC)

For better security, we can change the secret function for each different mobile agent, which implies that a compromised secret function does not decrypt other mobile agents. Even though a compromised secret function can decrypt other mobile agents (i.e., breaking only composite function), the malicious hosts still need secret keys for HES to decrypt sensitive data.

Another security issue that we should consider is how secure HES is. One thing quite sure is that a secret function with high degree and many terms will confuse untrusted hosts more than the one with low degree and less terms. Unfortunately, there is no guide-line in selecting a good secret function, but very complicated secret functions will hide many details of mobile agents from untrusted hosts while increasing the size of code. The weak (too simple) secret functions will provide faster performance, but less secure than the complicated ones in general.

One form of attacks to the function composition is decomposition attack to composite functions by malicious hosts. However, interestingly there is already a result on the hardness of decomposing a rational function (rational function is the quotient of two polynomial functions). According to Sander and Tschudin, Zippel argued that there is no polynomial time algorithm for decomposing multivariate rational functions [11, 16].

6.5 Strengths

Our approach can do many things that were not possible by many previously proposed approaches. It is an extension and implementation of mobile cryptography proposed by Sander and Tschudin [11, 13, 12]. Although they first proposed it, they did not suggest any practical idea of implementation; this is because they could not find or make an additive, multiplicative and mixed-multiplicative homomorphic encryption scheme to use in their idea.

One of the strengths of our approach is that ours allows encrypted agents to run without decryption.

The decryption of the encrypted result is exactly the reverse of the encryption; apply the inverse of the encrypting function first, and use the secret key of the HES to recover the actual result.

Another strength is that the encryption of inputs from malicious hosts is done automatically without the need for separate encryptions which cause the interruption of execution. The mixed-multiplicativity property of homomorphic encryption scheme (HES) in our approach encodes the encryption routine inside mobile agents for continuous computation. Additionally, the encryption time for each input is minimal, since it requires only a few lines of code with simple operations such as addition and multiplication; it calls a function, `split`, to randomly split an input, multiplies E(1) to each half, and adds each encrypted half.

The next strength is that our approach is safe from many privacy attacks. We can consider two types of attacks: attacks on code, and on data. The homomorphic encryption scheme of our approach protects data from attacks. The data in this context refers to any sensitive data such as credit card information, personal information, and any other important information that are very critical for decision making. The code privacy is obtained through our function composition. The code is segmented into several blocks through function composition, and none of the blocks depends on others for its execution, which causes confusion to malicious hosts. The three-address code is rewritten by our function composition technique to include some extra codes to confuse malicious hosts.

Another strength is that this approach prevents integrity attacks, which are of two types: attacks on code, and on data, including state information. The homomorphic encryption scheme, and the function composition technique used in our approach can prevent attacks on data including state information for mobile agents. Malicious hosts should have a clear idea of which variable or data to change to make a successful integrity attack (to change the decision so that it brings benefits to malicious hosts). For example, a malicious host wants to modify a mobile agent so that this mobile agent can be tricked into buying flowers from the malicious host. This requires the malicious host to know which data to change and where the data is located in the mobile agent. Our HES prevents malicious hosts knowing the value of a variable, because it is encrypted by HES.

Another strength can be found in the fact that we are extending Sander and Tschudin's idea of mobile cryptography, which has only an imaginary homomorphic encryption scheme with no actual encryption scheme. Furthermore, it limits the computation to some limited classes of mathematical functions (polynomial functions). However, our approach suggests a practical way of implementing the idea of mobile cryptography, and it is not limited to some mathematical functions; it can be applied to general functions within the limits of assumptions that stems from the property of ring theory.

The last strength is that our approach can protect the results from integrity and privacy attacks. From the definition of the homomorphic encryption scheme and function composition, it is very clear that the result generated from an encrypted mobile agent is automatically encrypted. This requires a decryption process for agent owners, but agent owners can expect the very same level of security for the results as identical to that of the encrypted mobile agent.

6.6 Weaknesses and Limitations

Our new approach can provide a range of security coverage broader than other security approaches for mobile agents; however, it does not provide perfect protection. We will discuss this in this section.

One of the biggest weaknesses is the code growth problem. As discussed already, our homomorphic encryption scheme adds a few extra codes for inputs from malicious hosts; furthermore, the FnC increases the size of the code. Although the size of the code is increased, the computing time will not increase proportionally, due to the fact that our homomorphism limits the possible operations to computationally cheap ones, such as addition and multiplication. There are also some compiler optimization schemes that we can use to reduce the size of the encrypted codes. There is no fixed mathematical expression or equation that best expresses the code growth and code reduction in our approach. This weakness will limit the application of our approach to the encryption of those functions that process sensitive data, while the rest of the program or mobile agent is left unencrypted. However, for a small mobile agent or

program, we can encrypt the whole code, data, and state information.

Another weakness of our approach is that the types of possible function calls in the encrypted function are somewhat limited. Basic input and output functions (`scanf` and `printf`) can be called without problems from encrypted functions or mobile agents. However, encrypted mobile agents or functions may have problems, if the encrypted mobile agents or functions try to call user-defined functions, or some other functions from a language library. For example, a user-defined function that generates a certain value (a hash value) from a function parameter may not work properly, as there is no guarantee that the hash value is correctly generated. However, this does not imply that all the user-defined functions cannot be called from an encrypted function. We believe that many user-defined functions and other pre-defined functions can be rewritten, or redefined carefully.

Another weakness is that there are some integrity attacks that cannot be tolerated by our approach. Our approach prevents integrity attacks if those attacks require some pre-analysis of code and data. For example, malicious hosts cannot change a decision making variable to trick mobile agents to take an action that benefits malicious hosts, because malicious hosts cannot know the values or decisions stored in the decision making variable. However, like many other security approaches, our approach is vulnerable to blind integrity attacks, which means that malicious hosts modify any parts of codes or data; this is a form of denial of service attack.

The next weakness can be found in the fact that our approach uses a homomorphic encryption scheme based on ring theory. This means that the possible types of operators are limited to addition and multiplication. In addition, it limits the data we can work on to integers modulo n. Furthermore, control structures in three-address code are not included in our approach.

Unlike some mobile agent security approaches such as PRAC [15], there is no mechanism for the detection of any modifications made to encrypted mobile agents and their results. Our approach does not provide complete protection against integrity attacks; thus it may be possible for a malicious host to modify an encrypted mobile agent, which would not crash, but would instead generate a wrong result. Some lucky integrity attacks, such as the one just mentioned, could go undetected, although the result would be incorrect and would mislead the agent owner.

In general, our approach suggests a practical idea of implementing mobile cryptography on a three-address code, with some limitations as explained in this section. From the limitations of our approach, we can summarize its possible applications: we can suggest that our approach could be used for mobile codes or programs that require integrity and privacy on code, data, and result. Our approach could also be used to encrypt functions, rather than the whole program, due to the code growth problem, unless the program is small. Unlike Sander and Tschudin's approach, the application of our approach is not limited to some mathematical classes of functions, but to general classes of functions.

7 Conclusion

In this paper, we have tried to address the problem of protecting mobile agents from untrusted hosts by proposing a new security approach which extends the idea of mobile cryptography originally proposed by Sander and Tschudin [13, 12]. Protecting mobile agents from malicious hosts is very hard due to the fact that hosts have complete control over any programs. We have reviewed many good approaches; however, they were somewhat less secure than we desire and do not provide both privacy and integrity to mobile agents and results. Sander and Tschudin argued that a homomorphic encryption scheme can encrypt a function, that the encrypted function can be evaluated without decryption, and that the result is automatically encrypted. Although there are many advantages of this approach, no practical method was proposed to implement this mobile cryptography. This is because Sander and Tschudin limited their proof to the use of additive, multiplicative, and mixed-multiplicative homomorphism to encrypt a few classes of mathematical functions, under the assumption that there exists homomorphic encryption scheme (they pointed out that they had found no such homomorphic encryption scheme exists as yet).

Our work, extending Sander and Tschudin's idea, provides an approach to implementing mobile cryptography. Our approach is a hybrid one, mixing a

homomorphic encryption scheme and function composition through which we can prevent privacy and integrity attacks on mobile agents and results. Even though our approach provides broader protections than other approaches, some attacks still cannot be prevented. This approach also has some limitations stemming from the organizing components such as ring theory basis of our homomorphic encryption scheme. However, our work still contributes to mobile agent security by solving some of the security problems that have been studied by many researchers. The summary of the specific contributions of this work are:

1. We proposed a practical method of implementing mobile cryptography by extending Sander and Tschudin's idea.
2. We developed a homomorphic encryption scheme, which is additively, multiplicatively, and mixed-multiplicatively homomorphic. The biggest problem of Sander and Tschudin's approach was that there have been no published homomorphic encryption schemes to use in mobile cryptography.
3. Our approach is not limited to a few mathematical classes of functions.
4. Our work allows the direct evaluation of encrypted mobile agents without any decryption.
5. The result generated from encrypted mobile agents are automatically encrypted, and only the agent owner can decrypt it.
6. It prevents many types of privacy attacks.
7. It prevents many integrity attacks, although it cannot prevent blind modification attacks, which are a type of denial of service.
8. Results from encrypted mobile agents have the same level of security protections as the mobile agents

In conclusion, this research work is an attempt to provide a broader range of protection for mobile agents, and it is hoped that this work can serve as a small contribution to the security of mobile agents.

8 Future Work

In this paper we proposed a hybrid approach, which is a combination of HES and FnC, and argued that ours can provide broader range of protection to mobile agents. However, there are some limitations and assumptions in our approach, which restrict the application of our approach and require further study. We discuss some of the future works that must be given considerations for the improvement of our approach as follows:

- Our modified encryption scheme is additive, multiplicative and mixed-multiplicative homomorphic encryption scheme. It is a simple cryptosystem, and requires extra work to develop more sophisticated encryption schemes with complete security analysis.
- Due to the assumption of ring theory, the possible operators are restricted to addition and multiplication only. If we can move up to field, we can add two more operators such as subtraction and division.
- The number sets that we are dealing with in our approach is integers, because of the assumption of ring theory. The number sets should be extended to some other types of numbering systems such as real numbers.
- The types of function calls within an encrypted function or mobile agent is limited to some primitive ones such as basic input and output. More study is required to find a way of calling user-defined and system functions within an encrypted mobile agent.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*, pages 462–512. Addison-Wesley, 1988.

[2] J. Domingo Ferrer and J. Herrera-í Joancomat. A privacy homomorphism allowing field operations on encrypted data. *Jornades de Matematica Discreta i Algorsmica*, 1998.

[3] Joshua D. Guttman and Vipin Swarup. Authentication for mobile agents. In *LNCS*, pages 114–136. Springer, 1998.

[4] Neeran Karnik. *Security in Mobile Agent Systems*. PhD thesis, Department of Computer Science and Engineering. University of Minnesota, 1998.

[5] Chandra Krintz. Security in agent-based computing environments using existing tools. Technical report, University of California, San Diego, 1998.

[6] Rainer Leupers. Lance: A compiler platform for embedded processors. *Embedded Systems/Embedded Intelligence*, 2001.

[7] Rainer Leupers. The lance v2.0, c compiler system. http://ls12-www.cs.uni-dortmund.de/lance/, 2001.

[8] Lars Rasmusson and Sverker Jansson. Simulated social control for secure Internet commerce. In *Workshop on New Security Paradigms*, pages 18–26, Lake Arrowhead, CA, September 1996.

[9] James Riordan and Bruce Schneier. Environmental key generation towards clueless agents. *Lecture Notes in Computer Science*, 1419:15–24, 1998.

[10] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–178, 1978.

[11] Tomas Sander and Christian Tschudin. Towards mobile cryptography. Technical report, International Computer Science Institute, Berkeley, 1997.

[12] Tomas Sander and Christian F. Tschudin. On software protection via function hiding. In *Information Hiding*, pages 111–123, 1998.

[13] Tomas Sander and Christian F. Tschudin. Protecting Mobile Agents Against Malicious Hosts. In Giovanni Vigna, editor, *Mobile Agent Security*, pages 44–60. Springer-Verlag: Heidelberg, Germany, 1998.

[14] Bennet Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

[15] Bennet Yee. A sanctuary for mobile agents. In *DARPA Workshop on Foundations for Secure Mobile Code Workshop*, March 1997.

[16] Richard E. Zippel. Rational function decomposition. In *In Proceedings of the International Symposium on Symbolic and algebraic Computation*, July 1991.