Mobile Agent Security Through Multi-Agent Cryptographic Protocols[†]

Stephen R. Tate
Department of Computer Science
University of North Texas
Denton, TX 76203, U.S.A.

Ke Xu
Department of Computer Science
University of North Texas
Denton, TX 76203, U.S.A.

***Abstract*—We consider the problem of keeping sensitive data and algorithms contained in a mobile agent from discovery and exploitation by a malicious host. The focus in this paper is on rigorous techniques based on cryptographic protocols. Algesheimer, Cachin, Camenisch, and Karjoth (*IEEE Security and Privacy, 2001*) devised a secure agent protocol in such a setting, where agents and hosts are mutually distrusting, but access to a "trusted third party" is available to all participants. In this paper, we present ways of removing the trusted third party, and achieving similar results through the application of multiple agents. As an agent on a remote host is trusted by neither the current host nor the agent originator, the remote agent cannot simply act as a "stand-in" for the trusted third party, and requires the design of non-trivial multi-agent protocols. In addition, our multi-agent protocol can proceed if any subset of the agents of a certain size is available at any particular time, adding fault-tolerance which did not exist in previous protocols, while achieving a high level of security. Our solution relies on well-tested cryptographic primitives, including threshold cryptography and oblivious transfer.**

***Keywords*—Internet Security, Mobile Agents, Cryptographic Protocols**

I. Introduction

Mobile agents are goal-directed, autonomous programs capable of migrating from host to host during their execution [1]. The combination of autonomy and mobility provides mobile agents enormous potential for application in today's Internet-based, distributed computing environment. Typical application areas, to name a few, include E-commerce, information retrieval, software distribution and administration, and network management. We consider the security issues of the mobile agent paradigm. From the security perspective, a mobile agent system consists of three basic components: the agent, the originator from which the agent starts (also known as the home host), and the remote hosts[1] that the agent visits. In a typical Internet application, there is little trust between the originator and the hosts, nor do the hosts trust each other. Security threats are therefore considered against the two main targets: either threats against the hosts, or threats against the agent, which represents the interests of the originator.

This paper addresses the attacks against mobile agents from malicious hosts. Because the host provides the executing environment for the agent, it has full control of the agent's code, data, and execution. Hence eavesdropping and alteration can easily be done to the agent and are hard to prevent. In fact, these kinds of attacks had been deemed by some people as impossible to prevent unless a secure, trusted hardware environment is available at the hosts [2]. While this statement may be true when taken literally, as demonstrated by this paper and some previous work, cryptography can make *meaningful* tampering with the agent impossible even in a software-only setting.

A. Security Definitions

In this section, we define the requirements of secure mobile agent computation, with a goal of achieving a sound and complete security mechanism that effectively prevents attacks against the agent as well as protects the host's data. The definitions we develop in this section improve on those in previous work in agent security, where definitions have been less precise and didn't offer the distinctions between various security objectives as done here.

An agent computation involves three objects, and we consider the privacy and integrity of each object independently: the agent code, the agent state (or the input to the agent that is transferred from its previous location), and the host data input. Privacy is defined in terms of the amount of information revealed to either the hosts or the agent originator, depending on the type of information involved. As static code can be signed by the originator, and host data is handled locally by the execution environment, integrity for these objects is easily obtained. However, we do need to consider integrity of the agent state; in other words, we would like to insure that the state resulting from a computation on a host is the correct answer that would be obtained if all parties acted honestly. For example, a re-computation attack, where the host repeatedly computes the agent function with different inputs until a favorable outcome is obtained, is a violation of state integrity.

In the most private and paranoid of settings, we would like all objects to achieve privacy and integrity goals: only the originator knows what the code/algorithm is; only the originator can decode the agent state; only the current execution host knows the data that it provides to the agent; and agent state cannot be falsified or manipulated to favor a particular host or entity. However, it is not difficult to see that achieving all of these

[†]This research is supported in part by NSF award 0208640.

[1] From now on, we'll simply use "hosts" when talking about the remote hosts.

	Agent State Privacy	Agent State Integrity	Agent Code Privacy	Limited Host Data Privacy	Verifiable Host Data Privacy	Complete Host Data Privacy
Dishonest Originator	N/A	N/A	N/A	1	2	
Dishonest Hosts	1&2	1&2	1			1&2

TABLE I
PRIVACY AND INTEGRITY ACHIEVED BY VARIANTS OF OUR PROTOCOL: 1=PRIVATE AGENT CODE; 2=PUBLIC AGENT CODE.

goals simultaneously is not possible, since privacy of the agent code means there is no guarantee that the agent doesn't simply hide the host data in its state and pass it back to the originator. In fact, since the host data is usually reflected in some way by the output state of the agent (after all, what is the purpose of the agent computation if it is not affected by the host data!), the issue of host data privacy is difficult to get a grasp on. To this end, we define three variations on host data privacy, with the amount of privacy increasing as the list progresses.

Definition 1: The amount of privacy afforded host data can be characterized in one of the following three ways.

- **Limited Host Data Privacy**: The amount of information revealed about the host data is limited only by the size of the agent state space.
- **Verifiable Host Data Privacy**: The agent function is known, so the amount of information leaked through the agent output state can be analyzed and verified.[2]
- **Complete Host Data Privacy**: No information about the host data can be obtained.

The first case, limited host data privacy, means we know nothing about the privacy afforded other than what is obvious from the external description of the agent function: if the host supplies 1024 bits of data to the agent, and yet the agent state is only 32 bits, the amount of information that can be passed on through the agent state is limited by this fact. On the positive side, limited host data privacy does imply that the protocol itself is secure, in the sense that it doesn't reveal any additional information beyond what is passed on through the agent state.

As explained above, we cannot achieve complete host data privacy with respect to the originator. At best, we can hope for verifiable host data privacy, so the information revealed may be assessed (and it might be the case that the information revealed is extremely low). On the other hand, we *can* hope for complete data privacy with respect to other hosts in the agent system — in other words, while the originator may learn some information about a host's data, none of the other hosts should be able to discover any such information.

Note that we do not address the availability part of the standard Confidentiality/Integrity/Availability trio of security goals. A host can simply destroy an agent once it is received, resulting in a denial-of-service attack. However, we do note that a related attack, that of keeping the agent on the existing host and simulating a different host[3], can be stopped by adding some accountability to the data. In particular, a host signs its data (or an authentication token) when it executes the agent code.

B. Our Result — A Multi-Agent Protocol

We present a software-only multi-agent protocol for secure mobile agent computation, with two variations: either the agent function is known, or it is kept secret. Table I summarizes the security achieved by our protocols according to the definitions in the previous section. Both variants provide for agent state privacy and integrity, as well as complete host data privacy with respect to other hosts. If the agent carries an unknown function, then it also achieves code privacy and limited host data privacy (with respect to the originator). If the agent carries a known function, and a non-interactive zero-knowledge proof that the encrypted circuit contained within the agent implements that function, then it achieves verifiable host data privacy with respect to the originator.

Our work follows a recent direction of applying cryptographic protocols in mobile agent computation which was initiated by Sander and Tschudin in 1998 [3]. They proposed the use of a homomorphic encryption scheme in order to allow secure computation on untrusted hosts. However, their result had the drawback that it was only applicable to a limited set of functions (the set of rational functions, to be precise). Following work has expanded the range of securely computable functions, eventually to all polynomial-time functions [4],

[2] Another way of saying this is that no information about the inputs can be learned other than what follows from the outputs, which is typically the goal in cryptographic protocols and in previous work on the agent security problem.

[3] For example, instead of sending a bidding agent to the next bidder, the host keeps it locally and provides the next bid itself.

[5], [6]. Based on Yao's general purpose, two-party secure function evaluation (SFE) protocol [7], the protocol of Algesheimer, Cachin, Camenisch, and Karjoth [6] (hereafter referred to as the ACCK protocol) provides privacy and integrity for the agent code and state, as well as complete host data privacy with regard to other hosts. Depending on whether the agent code is private or public, the privacy of host data with regard to the originator can be either limited or verifiable. Unfortunately, this protocol requires the participation of a trusted third party who doesn't collude with either the originator or any host. Although this may be a reasonable assumption in some circumstances, our work improves upon the ACCK protocol by eliminating this trusted third party, resulting in a secure protocol for the standard mobile agent paradigm through the use of multiple agents and two cryptographic tools: distributed threshold decryption and oblivious transfer. The introduction of multiple agents also provides additional benefits such as protection against collusion, fault tolerance, and parallelism, as explained in the next section. Note that all previous work on this problem used a model with only a single agent traveling from host to host, and while this could be put into a multi-agent setting as a set of independent and non-interacting individual agents, it wouldn't take advantage of the multi-agent setting, especially the possibility of cooperation among agents.

As described later, the general techniques (both in prior work and in this paper) base computation on simulating boolean circuits in an encrypted manner. While representing functions at such a low level as boolean circuits is expensive, only the privacy-critical part of the agent code needs to be implemented in circuits and be evaluated following our protocol. The rest of the agent should still be in a traditional executable form (which, for instance, could be machine code, Java byte code, or Perl source code). For example, an agent collecting bids for a good or service may wish to perform a "min" operation on the current lowest bid and the bid from the current host, remembering the lower of the two bids. The lowest bid is part of the agent state, and we would like to keep this value private and unknown to all hosts that execute the agent. This can be accomplished by a simple encrypted circuit, which could be represented using less than 9k bytes of data (this figure was reached with a basic, unoptimized circuit, and a moderate security parameter of 80-bits per encrypted boolean value). While using 9k bytes to keep 32 bits secret is certainly a very large overhead, 9k seems to be a reasonable value for many practical settings.

C. Advantages over the Previous Work

Besides meeting the security goals described in the previous section, our protocol has the following advantages over previous work.

- *Eliminating the requirement of a trusted third party.* One of the main goals of cryptographic protocols is to enable secure computation in an insecure environment. The weaker trust assumption a protocol has, the more valuable it is. In the ACCK protocol, the trusted third party plays an essential role in maintaining security, making it the single point of failure and the performance bottleneck. Our protocol eliminates this requirement, and shows that by using multiple agents, it is possible to attain the same degree of security without the participation of any trusted third party.
- *Protection against collusion.* We employ *threshold cryptography* to add protection against collusion of malicious hosts. The protocol is created using a safety parameter s which defines a *threshold* $t = \lceil \frac{n+s}{2} \rceil$, where n is the number of agents. Any subset of colluding hosts of size smaller than s cannot get any unauthorized information about the protected computation, while any subset of size t or more can complete the necessary computation. The gap between s (the number of colluding hosts the protocol is safe from) and t (the number of hosts required to complete the protocol) seems to be unavoidable, and is addressed in the "Analysis" section later in the paper. The safety parameter s leaves room for the user to select the safety appropriately for a particular application and obtain the proper balance between security and efficiency.
- *Fault-tolerance.* In addition to protection against collusion, threshold cryptography adds an element of fault tolerance. If the threshold t is less than the number of active agents, then any subset of t agents can complete any round of the protocol. If some agents have disappeared, or are temporarily unavailable, the computation can still proceed as long as t agents are available. Thus, as long as $t < n - 1$, there is no single point of failure, such as the trusted third party in the ACCK protocol.
- *Parallelism.* Our work is the first to look specifically at the multi-agent setting for agent security. While the parallelism we achieve is not a specific attribute of our protocol, by specifically considering a multi-agent setting we reap the benefits of traditional multi-agent advantages, including parallelism.

D. Organization of the Paper

Section II presents a general model for the agent computation. In Section III, we review the cryptographic tools used in this paper and present our protocol. A brief analysis is also provided. Section IV gives the conclusion.

II. The Multi-Agent Model

The work of Algesheimer et al. [6] provides a good definition of the mobile agent model, which we extend here for the multi-agent setting.

The participants in the computation are an originator O, n mobile agents $\mathcal{MA}_1, \ldots, \mathcal{MA}_n$, and ℓ hosts $H_1, \ldots, H_\ell$. The hosts are divided into n disjoint subsets. Each agent visits exactly one subset of hosts, doing some computation at one host, updating its current state and producing some output to the host if required by the application, then migrating to the next host. In the end, all n agents return to the originator and their partial computation results are combined to get the final result.

More concretely, we let $\mathcal{X}$ be the set of possible states of the agents, $\mathcal{Y}$ be the set of possible inputs from the hosts, and $\mathcal{Z}$ be the set of possible outputs that are given to the host. Then agent $\mathcal{MA}_i$, executing on host $H_{(i,j)}$ of subset i, computes two functions

$$g_{(i,j)} : \mathcal{X} \times \mathcal{Y} \rightarrow \mathcal{X} \qquad \text{and} \qquad h_{(i,j)} : \mathcal{X} \times \mathcal{Y} \rightarrow \mathcal{Z} \ ,$$

where $g_{(i,j)}$ is the "agent state update" function that produces the state for input to the next host the agent visits, and $h_{(i,j)}$ is the function that provides output for $H_{(i,j)}$. The original agent state, set by the originator, is denoted by $x_{(i,1)}$, and $x_{(i,j+1)}$ is the state that results from the computation at host $H_{(i,j)}$. Additionally, we let $y_{(i,j)}$ denote the private input of $H_{(i,j)}$, and $z_{(i,j)}$ denote the output produced for $H_{(i,j)}$. The functions then satisfy the following two requirements:

$$x_{(i,j+1)} = g_{(i,j)}(x_{(i,j)}, y_{(i,j)})$$

$$z_{(i,j)} = h_{(i,j)}(x_{(i,j)}, y_{(i,j)}) \ .$$

III. The Multi-Agent Protocol for Secure Agent Computation

In this extended abstract, we present our protocol in a mostly brief and informal manner. In particular, we only informally describe the cryptographic tools employed in this protocol, and we omit the rigorous proof of security of the protocol, leaving that to a separate paper.

A. Cryptographic Basis and Tools

In this and the following section, we lay the basic cryptographic groundwork needed for our protocol. In this section we first give an informal definition of oblivious transfer, an important cryptographic primitive, and then describe the basics of encrypted circuits and the ACCK protocol.

Definition 2: **1-out-of-2 Oblivious Transfer**: A has two binary strings s_0 and s_1. B has a bit b. After executing the protocol, B receives s_b, but doesn't learn anything about s_{1-b}, while A doesn't learn b.

The ACCK protocol is based on Yao's two-party SFE protocol [7], which is for computing any two-party function securely in the sense that after the computation each party only learns the output it is entitled to, but nothing else. Since the agent computation can be modeled as evaluating a series of two-party functions $g_{(i,j)}$ and $h_{(i,j)}$ between the originator (whose input is in the form of agent state) and each of the hosts, Yao's protocol certainly applies here. The core idea in Yao's protocol is the so called *encrypted circuit*. An encrypted circuit is a boolean circuit where the signals on the wires are random strings whose meanings (i.e., 0 or 1) are hidden from the evaluator. Accordingly, the truth tables of the gates are mappings from the input signals to the output signals which obey the corresponding semantics of these strings and the functionality of the gates. In the two-party protocol, party A creates an encrypted circuit to compute the desired function. Then A sends to B the encrypted circuit, the semantics of the signals on the output wires for B's entitled output, and the signals corresponding to A's input for the circuit. Since the semantics of A's input signals are hidden from B, B cannot find out A's input. For B to receive the signals which correspond to its own input, A engages with B in an 1-out-of-2 oblivious transfer so that B gets only the correct signals for its input bits but not the complements, while A doesn't learn B's input. Having both parties' input signals, B evaluates the encrypted circuit and gets the output signals for both parties. Then it recovers its own output using the semantics definition of its output signals, and sends back A's output which is still in the form of signals. Additional protection using input commitment, collaborative random bit generation, and zero-knowledge proofs is required for preventing each party from deviating from its prescribed behavior. A detailed description of the encrypted circuit can be found in [8], but the corrected version from [9] should be used to avoid a flaw in the earlier technique.

In the ACCK protocol, the originator O plays the role of A and the hosts play the role of B, and the agent code is the encrypted circuit. Yao's protocol requires interaction between each of pair of A and B, which is impossible for the agent setting as O is assumed offline after sending out the agents. Their solution is to have O encrypt all the signals (for both 0 and 1) for the input wires corresponding to the hosts' inputs using a trusted party T's public key, and let the agent bring the encrypted signals to the hosts. Before executing the agent, the hosts select the encrypted signals for their inputs and ask T to decrypt the chosen signals. Essentially this process implements the oblivious transfer using the trusted third party.

We show that the trusted third party T can be eliminated from the protocol. Instead, we use multiple mobile agents to decrypt the encrypted signals for the hosts' inputs. Since every agent is at risk of being corrupted by a malicious host, we cannot trust any particular agent to perform its decryption task honestly, which is, it only decrypts one signal for each bit of a host's input, not the complement, and it doesn't learn the requesting host's input data. Therefore, in the next section we define a new cryptographic primitive called *Oblivious Threshold Decryption*, and show how it can be realized using known techniques for non-interactive threshold decryption and 1-out-of-2 oblivious transfer.

B. Oblivious Threshold Decryption

Threshold cryptography plays a key role in our multi-party protocol, so we give a basic definition here.

Definition 3: **Threshold Cryptosystem**: In a t-out-of-n threshold cryptosystem, there are n parties. There is a single public key, while the private key is shared among the n parties in such a way that decryption is easy for any subset of t or more parties working together, but is not feasible for less than t parties.

Threshold cryptography has been extensively studied, and good solutions exist, such as the technique of Shoup and Gennaro [10]. However, we need something a little more involved than this, so we define a new cryptographic primitive called "Oblivious Threshold Decryption" that meets our needs.

Definition 4: **Oblivious Threshold Decryption (OTD)**: Parties $A_1, A_2, \ldots, A_n$ hold shares of a decryption function, as in the standard threshold cryptosystem of Definition 3. Party A_1 has a pair of encrypted binary strings $(\overline{s}_0, \overline{s}_1)$ and a private bit b, and takes place in a distributed computation with at least $t-1$ of the other parties so that she gets the decrypted value of $\overline{s}_b$ without learning any information about the decrypted value of $\overline{s}_{1-b}$, while the other hosts learn no information about the value of A_1's selection bit b. If fewer than $t-1$ other hosts participate, then neither $\overline{s}_0$ nor $\overline{s}_1$ can be decrypted.

Our multi-agent protocol will use OTD as its foundation (in addition to encrypted circuits), and the main purpose of OTD is to protect against collusion of bad agents (by virtue of the threshold properties), and to prevent a host from getting signals other than those corresponding to its own input or a bad agent from learning the host's input (by virtue of the oblivious transfer properties). OTD can be built up using known techniques for non-interactive threshold decryption and 1-out-of-2 oblivious transfer. In particular, the following realization of OTD is based on the protocols of Shoup and Gennaro [10] and Bellare and Micali [11].

Algorithm 1: OTD-PROTOCOL: Let TC refer to a t-out-of-n threshold cryptosystem, and assume there is a trusted dealer D that can create keys for TC. Let $A_1, \ldots, A_n$ denote the n parties in the system. Then we can implement OTD using the following steps:

1. OTD-SETUP: D uses the *Key Generation* algorithm for TC to create a single public key PK, a verification key VK, and n private key (or "secret key") shares SK_i, for $i = 1, \ldots, n$. D distributes private key share SK_i to party A_i.

2. OTD-DISTRIBUTE: Party A_1 has a bit b and two strings $\overline{s}_0 = E_{PK}(s_0)$ and $\overline{s}_1 = E_{PK}(s_1)$, where $E_{PK}(\cdot)$ denotes the *Encryption* algorithm of TC using PK as the encryption key. To obtain the decrypted string s_b while keeping b secret, A_1 sends both $\overline{s}_0$ and $\overline{s}_1$ to (at least) $t-1$ other parties, $A_{i_2}, \ldots, A_{i_t}$. For uniformity of notation, we set $i_1 = 1$, so that the set of t parties participating in the computation is $\{A_{i_1}, \ldots, A_{i_t}\}$.

3. OTD-SHARE-CREATION: Each of the t parties checks the identity of $\overline{s}_0$ and $\overline{s}_1$ through TC's *Label Extraction* algorithm, and determines the validity of A_1's request. If a party A_{i_j} determines that the request should not be honored, then A_{i_j} does nothing; therefore, if no more than $t-1$ servers honor the request, the decryption cannot be (even partially) done. Otherwise, A_{i_j} applies TC's *Decryption* algorithm on $\overline{s}_0$ and $\overline{s}_1$, and obtains its corresponding decryption shares $s_0^{i_j}$ and $s_1^{i_j}$.

4. OTD-SHARE-COMBINATION: A_1 engages with each of the $t-1$ other parties in a 1-out-of-2 oblivious transfer, and receives only the $t-1$ decryption shares of $\overline{s}_b$, without revealing b to the other parties. If a party does not provide a share because it detected cheating in the previous step, then the protocol cannot proceed. A_1 then combines it's own decryption share of $\overline{s}_b$ with the $t-1$ shares it receives in order to form the required set of t decryption shares. Using TC's *Share Verification* algorithm, A_1 checks the validity of the t decryption shares. If all are valid, A_1 combines them with the *Share Combination* algorithm to obtain s_b.

While this algorithm uses a "Trusted Dealer", this is not a trusted third-party in the agent protocol — in fact, the originator plays the role of the trusted dealer when this protocol is used in the agent setting. Also note that while the protocol seems to require several rounds of communication as it is described above (since the oblivious transfer is put in the middle of the threshold decryption), we can combine messages so that this is a one-round OTD protocol: the message from A_1 in the OTD-DISTRIBUTE step can be combined with the first message of the non-interactive 1-out-of-2 oblivious transfer in the OTD-SHARE-COMBINATION step. Then the single response message from each party A_{i_j} in the oblivious transfer is the only other communication required.

The security goals of OTD, which are for A_1 to learn only one decrypted string and no other party to learn which string A_1 has recovered, follow easily from the security of the underlying threshold cryptosystem and the 1-out-of-2 oblivious transfer protocol.

C. The Multi-Agent Protocol

Our protocol assumes a typical multi-agent setting but requires that each mobile agent visits a disjoint set of hosts. Assume there are ℓ hosts and n agents in the system, where $n \leq \ell$. Let s be the *safety* of our protocol. That is, the protocol is secure against collusion of up to $s-1$ hosts. Without loss of generality, assume the agent state is n_x bits, the input from each host is n_y bits, and the output for each host is n_z bits. An encrypted circuit is a tuple $(\mathcal{C}, \mathcal{L}, \mathcal{K}, \mathcal{U})$, where $\mathcal{C}$ is the encrypted form of the circuit, $\mathcal{L} = ((L_{1,0}, L_{1,1}), \ldots, (L_{n_x,0}, L_{n_x,1}))$ and $\mathcal{K} = ((K_{1,0}, K_{1,1}), \ldots, (K_{n_y,0}, K_{n_y,1}))$ are the lists of input signals for the agent state and the host input, respectively, and $\mathcal{U} = ((U_{1,0}, U_{1,1}), \ldots, (U_{n_z+n_x,0}, U_{n_z+n_x,1}))$ is the list of output signals to the host as well as the signals for the updated agent state. Each pair of signals are associated with one bit of input or output, with the first signal of the pair encoding 0 and the second one encoding 1. Notice that the updated agent state, $((U_{n_z+1,0}, U_{n_z+1,1}), \ldots, (U_{n_z+n_x,0}, U_{n_z+n_x,1}))$, equals the $\mathcal{L}$ of the encrypted circuit for the next host.

The basic framework of our protocol is the same as the ACCK protocol, but the decryption of the signals corresponding to the hosts' inputs is done by the mobile agents through the OTD protocol. The following is a step-by-step description of our protocol.

1. It can be proved (see the next section) that the threshold t of the cryptosystem must be at least $\lceil \frac{n+s}{2} \rceil$ to insure security against $s-1$ colluding hosts, each controlling an agent. The originator O executes step OTD-SETUP with parameters t and n to generate a public key PK, a verification key VK, and n private key shares SK_i.
2. O divides the ℓ hosts into n disjoint subsets. In the following, we identify a host with a pair (i,j), where i is the subset id and j is the host id within that subset. For each host $H_{(i,j)}$, O creates an encrypted circuit $(\mathcal{C}^{(i,j)}, \mathcal{L}^{(i,j)}, \mathcal{K}^{(i,j)}, \mathcal{U}^{(i,j)})$ with $\mathcal{C}^{(i,j)}$ being the agent code to be executed on that host, implementing the agent functions $g_{(i,j)}$ and $h_{(i,j)}$. In addition, to indirectly communicate the private input signals to host $H_{(i,j)}$, O encrypts $\mathcal{K}^{(i,j)}$ and its label (i,j) with the Encryption algorithm using PK. Denote the encrypted signals as $\overline{\mathcal{K}}^{(i,j)} = ((\overline{K}^{(i,j)}_{1,0}, \overline{K}^{(i,j)}_{1,1}), \ldots, (\overline{K}^{(i,j)}_{n_y,0}, \overline{K}^{(i,j)}_{n_y,1}))$.
3. Next, O creates n mobile agents. For agent $\mathcal{MA}_i$, $i = 1, \ldots, n$, its code includes the encrypted circuits $\mathcal{C}^{(i,j)}$, the encrypted signals $\overline{\mathcal{K}}^{(i,j)}$, and $((U^{(i,j)}_{1,0}, U^{(i,j)}_{1,1}), \ldots, (U^{(i,j)}_{n_z,0}, U^{(i,j)}_{n_z,1}))$, for all the hosts in subset i. O also sets $L'^{(i,1)}_k = L^{(i,1)}_{k,x_{(i,1),k}}$, for $k = 1, \ldots, n_x$, as the initial state of $\mathcal{MA}_i$. In addition, $\mathcal{MA}_i$ is assigned a private key SK_i, and O also signs all these components of $\mathcal{MA}_i$ with its private key (this is not the private key used in the threshold decryption, but rather a normal, static private key whose corresponding public key is certified and known by the hosts). After this, O sends out $\mathcal{MA}_i$ to $H_{(i,1)}$, the first host of subset i, for $i = 1, \ldots, n$. This concludes O's initial work, and O can now go offline until the last step.
4. Every host $H_{(i,j)}$, upon receiving agent $\mathcal{MA}_i$, first checks the signature of O. Then it locates at least t agents including $\mathcal{MA}_i$ (recall that t is the threshold), and executes step OTD-DISTRIBUTE using the signed list of encrypted signals $\overline{\mathcal{K}}^{(i,j)}$.
5. Each of the t agents checks the signature of $\overline{\mathcal{K}}^{(i,j)}$. Then it extracts the label (i,j), and checks its decryption history to see if this list of encrypted signals has been decrypted before. If either the signature is invalid or the list has been decrypted before, the agent detects cheating and refuses to decrypt the list. Otherwise, the agent executes step OTD-SHARE-CREATION to obtain the decryption shares for all pairs of encrypted signals of $\overline{\mathcal{K}}^{(i,j)}$.
6. The t agents (including $\mathcal{MA}_i$) execute step OTD-SHARE-COMBINATION and as a result host $H_{(i,j)}$ learns the decrypted input signals $\mathcal{K}^{(i,j)}$ that correspond to its private input $y_{(i,j)}$.[4]
7. $H_{(i,j)}$ evaluates the encrypted circuit $\mathcal{C}^{(i,j)}$ using the input signals that it now knows, decrypts its own private output using the corresponding $\mathcal{U}^{(i,j)}$ vector, and sends the agent (consisting of the encrypted circuits and signals for the following hosts as well as its updated state in the form of signals) to the next host in the subset, or back to the originator if this was the last host in the subset. If the transfer was to a different host, that host repeats steps 4 – 7.
8. Finally all n agents return back to O with their final states. O decrypts these signals to get the values they represent and combines the results into the final result.

D. Analysis

Of all the security goals we list in Table I, the privacy and integrity of agent state and code as well as complete host data privacy with regard to dishonest hosts are guaranteed from the security of Yao's protocol and the OTD protocol, as long as all parties follow the protocol honestly. Essentially, the OTD protocol implements the oblivious transfer from the originator to the

[4] Note that nothing requires $H_{(i,j)}$ to use it's "real" input for $y_{(i,j)}$, but rather could pretend that it's input was different, obtaining different decrypted input signals. However, these signals would correspond to *some* valid private input, which we then take to be the input that host $H_{(i,j)}$ has committed to.

hosts. Therefore, the agent computation realizes Yao's protocol and hence secure function evaluation between the originator and the hosts.

"Cheating" by not following the protocol is protected against through the use of signatures on the encrypted labels, and the security of the underlying primitive operations. One point that bears further exploration is the possibility of a host H using two different subsets of hosts in different OTD executions in order to decrypt more than one set of input signals, thereby compromising the integrity of the agent state. For this to work, any overlap between these two sets of hosts must consist of only bad hosts which collude with H, or otherwise the cheating would be caught (and stopped) in step 5 when the good hosts check their decryption histories. However, the following lemma shows that this is impossible.

Lemma 1: For any two subsets S_1 and S_2 of size t, $S_1 \cap S_2$ contains at least one good host.

Proof: S_1 and S_2 must each contain $t - |S_1 \cap S_2|$ unique hosts, as well as the $|S_1 \cap S_2|$ shared hosts, so the total number of hosts required is at least $2(t - |S_1 \cap S_2|) + |S_1 \cap S_2| = 2t - |S_1 \cap S_2|$.

Assume for the sake of contradiction that the lemma statement is not true, so $S_1 \cap S_2$ consists of only "bad" hosts. Because the safety parameter bounds the number of bad hosts, $|S_1 \cap S_2| \leq s - 1$, and so the total number of hosts is at least

$$\begin{aligned} 2t - |S_1 \cap S_2| &\geq 2t - (s-1) \\ &\geq 2\left\lceil \frac{n+s}{2} \right\rceil - s + 1 \\ &\geq n + s - s + 1 \\ &= n + 1 \end{aligned}$$

which contradicts the fact that we only have n hosts. Therefore, at least one of the hosts in $|S_1 \cap S_2|$ must be good. ■

IV. Conclusion

We present a software-only, multi-agent protocol for protecting mobile agents against tampering by malicious hosts. Compared with the prior work on this issue, our protocol satisfies the same security requirements but with the weakest possible assumptions (nothing more than a standard mobile agent setting), and maintains the most general functionality achieved so far: capable of computing any polynomial-time function. In particular, we have removed the reliance on a trusted third party that existed in the previous protocol, and maintained security even in a setting with a multitude of untrusted hosts. Our protocol also adds fault-tolerance and parallelism to the previous results.

References

[1] J. M. Bradshaw, "An introduction to software agents," in *Software Agents*, J. M. Bradshaw, Ed., chapter 1, pp. 3–46. AAAI Press, 1997.

[2] D. Chess, B. Grosof, C. Harrison, D. Levine, and C. Paris, "Itinerant agents for mobile computing," *IEEE Personal Communications*, vol. 2, no. 5, pp. 34–49, Oct. 1995.

[3] Tomas Sander and Christian F. Tschudin, "Protecting mobile agents against malicious hosts," in *Mobile Agents and Security*, G. Vigna, Ed., vol. 1419 of *Lecture Notes in Computer Science*, pp. 379–386. Springer Verlag, 1998.

[4] Tomas Sander, Adam Young, and Moti Yung, "Non-interactive cryptocomputing for NC^1," in *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999, pp. 554–566.

[5] Christian Cachin, Jan Camenisch, Joe Kilian, and Joy Müller, "One-round secure computation and secure autonomous mobile agents," in *Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP)*, U. Montanari, J. P. Rolim, and E. Welzl, Eds. 2000, vol. 1853 of *Lecture Notes in Computer Science*, pp. 512–523, Springer Verlag.

[6] Joy Algesheimer, Christian Cachin, Jan Camenisch, and Günter Karjoth, "Cryptographic security for mobile code," in *Proc. IEEE Symposium on Security and Privacy*, May 2001, pp. 2–11.

[7] Andrew Chi-Chih Yao, "How to generate and exchange secrets," in *Proc. 27th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1986, pp. 162–167.

[8] Phillip Rogaway, *The Round Complexity of Secure Protocols*, Ph.D. thesis, Laboratory for Computer Science, MIT, April 1991.

[9] Stephen R. Tate and Ke Xu, "On garbled circuits and constant round secure function evaluation," 2003, UNT CoPS Lab Technical Report 2003–2. Available at `http://cops.csci.unt.edu/`.

[10] Victor Shoup and Rosario Gennaro, "Securing threshold cryptosystems against chosen ciphertext attack," *Journal of Cryptology*, vol. 15, no. 2, pp. 75–96, Spring, 2002, A preliminary version appears in the proceedings of EuroCrypt'98.

[11] Mihir Bellare and Silvio Micali, "Non-interactive oblivious transfer and applications," in *Advances in Cryptology — Crypto'89 Proceedings*, Gilles Brassard, Ed. 1989, vol. 435 of *Lecture Notes in Computer Science*, pp. 547–557, Springer Verlag.