System Design and Implementation of Secure Mobile-Agent Computation with Threshold Trust

Henry Lien
CS490a Senior Project
Yale University
Computer Science Department
Yang Richard Yang, Advisor

December 13, 2002

Abstract

This paper focuses on an implementation of a portion of a system that can protect mobile agents from potentially malicious hosts. In this scheme, mobile agent computations are partitioned into two parts, a security-sensitive portion and a non-security-sensitive portion. The security sensitive portion is encoded in a garbled circuit and access to the translation tables for the circuit is controlled through the use of the verifiable distributed oblivious transfer protocol (VDOT). This protocol requires hosts to communicate with a group of servers in order to access private information. If any server among this group cheats, it can be detected, as long as more than a threshold of servers do not collude. Evaluation of the costs of this system shows that it is a usable system.

1 Introduction

The mobile agent is a new computational paradigm that can give people greater access to resources on the Internet [1, 2, 3]. By delegating tasks to mobile agents, users can finish extended or complicated tasks that they would rather not or cannot perform themselves. Moreover, sending mobile agents close to information sources can reduce network traffic and application latency.

The success of the mobile agent paradigm depends on security. Specifically, a mobile agent system should be able to protect both the hosts on which visiting mobile agents run and the mobile agents that carry security-sensitive information. In the past, the focus of mobile-agent security has been on protecting the safety and
the integrity of visited hosts. To achieve this objective, researchers have proposed novel techniques such as the Sandbox architecture [6], which restricts the access of a visiting mobile agent, and proof-carrying code [8], which allows a host to efficiently verify that the visiting mobile agent will not do harm to the host.

Protecting the host against malicious mobile agents is not enough, however. The security of the mobile agent paradigm also depends on the protection of mobile agents against malicious hosts, who might benefit from peeking into the agent’s private computation. Take, for example, the following situation: a college student who can only use lab computers needs to search for an airline ticket. Since the search may take a while and the student cannot lock a lab computer, she wants to send a mobile agent on her behalf to visit different airlines. At each site, the agent will query the available flights and update the best flight it has found so far. Since good deals come and go, the student decides that, if the price of a flight is below a threshold, the agent should book the flight at once on her behalf. It is clear that a host has incentive to attack such an agent. For example, the host may want to ensure that its flight is recorded as the best flight, even though there is another airline providing a lower price. The host might also want to discover the agent’s price threshold. By knowing this threshold, the host can make a higher profit by offering a price just below the threshold, while normally the host would offer a much lower price if it did not know this private information.

Previous research shows that protecting mobile agents from such malicious hosts can be achieved by special-purpose hardware [12, 14]. Surprisingly, Sander and Tschudin [9] recently pointed out the possibility of a software solution. Motivated by this result, a few other software-only solutions have been proposed, e.g., [10, 4, 7]. In particular, Algesheimer, Cachin, Camenisch and Karjoth [4] propose a general model and solution to the mobile-agent-security problem. Figure 1 shows the architecture of their scheme, which we refer to as the ACCK scheme. In the ACCK scheme, the originator encrypts its secret material using the public key of a trusted server. When the agent visits a host, the host interacts with the trusted server to gain the material. Although the ACCK scheme may work well under some scenarios, the security of this scheme relies on a single trusted server. Thus, if this trusted server is corrupted, the privacy of both the originator and the hosts can be compromised. Consider the airline-ticket agent. The host of an airline may corrupt the system’s trusted server and thus take advantage of all the users who use this trusted server. Therefore the entire system can be compromised by corrupting a single server.

This paper discusses a mobile agent system that does not suffer from the above

---

1Another possibility is that the originator first sends its security related material to the trusted server.
A design and implementation of the key components of this scheme are evaluated for performance. This is likely the first effort to implement a system that protects the privacy of mobile agents. To write an agent in this system, the designer extracts the security-sensitive portion of the agent into a function. Then the function is encoded as a garbled circuit, which is carried by the agent. Since one only applies the security mechanism to the security-sensitive portion of an agent, the system is efficient. Since the result of the security-sensitive portion is interpreted by the normal portion of the agent, all that a host needs to provide is an interpreter of garbled circuits. As a result, the system provides a general-purpose solution. Measurements of the performance of the system show that the overhead is acceptable. Thus, protecting mobile agents is not only possible, but also can be implemented efficiently.

The rest of this paper is organized as follows. Section 2, discusses a model of mobile-agent computation and its security requirements. Section 3 reviews related primitives. Section 4 presents the system architecture and specifies the protocol, followed by an evaluation of the implementation and performance of the system in Section 5. Section 6 presents the conclusions that can be drawn from the performance evaluation.

Figure 1: The ACCK Scheme

weakness. In particular, by applying the recent result, verifiable distributed oblivious transfer (VDOT) [15], the single trusted server is replaced with a group of servers. By distributing trust among a group of servers, this system strengthens the security of the ACCK scheme by increasing the number of servers that need to be corrupted before the system is corrupted.

2The garbled circuit interpreter can also be carried by the agent, in principle. However, requiring each host to provide a garbled circuit interpreter can greatly reduce the communication overhead.
2 Model and Security Requirements

2.1 A Model of Mobile-Agent Computation

In the model of mobile-agent computation discussed in this paper, which extends that proposed by Algesheimer, Cachin, Camenisch, and Karjoth in [4], the originator generates a mobile agent and sends it to the first host; each host runs the agent and then forwards it to the next host; the last host sends the agent back to the originator.

When the mobile agent runs at each host, it has two objectives: 1) update its internal state; and 2) compute a local output to the host. Both the state update and the local output depend on the local input of the host and the state of the agent when it arrives at the host.

In the airline-ticket agent example, the state of the agent is the threshold price and the current best flight with its price. At each host, the local input to the agent is the offered flight and price at the host; the local output of the agent is whether or not the offered price is below the agent’s threshold, which implies whether or not the agent will book the flight at once. As a result of running the airline-ticket agent at a host, the agent will also update its current best flight.

An implementation of the airline-ticket agent is shown in Figure 2 and Figure 3. In this implementation, the general portion of the agent queries the local database and then invokes the function update to make the security-sensitive update. The local input from the host to the update function is offered_price and offered_flight. The output to the local host is commit. Note that the update function stores the private state of the agent — the variables best_price, best_flight and threshold.

2.2 Security Requirements

Rather than discuss all security requirements that arise in the model just presented, this paper focuses primarily on privacy requirements. Readers who are interested in other security requirements such as integrity should look at [11]. The security requirements focus on here are as follow:

- (Originator’s Privacy) The security-sensitive portion of a mobile agent is private against the hosts and other involved parties. This requirement is obvious in the sample airline-ticket agent.

- (Hosts’ Privacy) The local input and output at a host are private against the originator of the agent and other involved parties. Although our airline-ticket
AIRLINE-TICKET-AGENT($date, src, dst$)

▷ date: departure and return date of the flight
▷ src: source of the flight
▷ dst: destination of the flight

Search local database for offered_flight and offered_price at the given date from src to dst;
commit = update(offered_flight, offered_price);
if (commit)
   print("book your flight");
else
   print("pending review by the user");

Figure 2: An Airline-Ticket Agent

UPDATE($offered-flight, offered-price$)

▷ local input: offered-price, offered-flight
▷ state: best-price, best-flight, threshold
▷ local output: commit

if (offered-price $threshold)$
   commit = true; // accept the flight
go back to originator;
else
   commit = false; // pending
   if (offered-price $best-price$) {
      best-price = offered-price;
      best-flight = offered-flight;
   }

Figure 3: The Update Function of the Airline-Ticket Agent

example does not impose this requirement, it is easy to come up with examples that need this requirement. For example, the query of an agent on a host may depend on the private data of the host, and therefore the local input and
output should be kept private from others.

- (No Single Trusted Server) There should not be any single trusted server that can compromise the privacy of the originator or the hosts. This requirement is discussed in the Introduction.

- (Cheater Detection) If any server involved in a computation cheats, the cheater should be detected.

3 Garbled Circuits and Verifiable Distributed Oblivious Transfer

The system described in this paper requires an understanding of garbled circuits and verifiable distributed oblivious transfer (VDOT), the two fundamental building blocks of this system.

3.1 Garbled Circuits

To protect the privacy of a mobile agent, one converts a security-sensitive function of a mobile agent into a garbled circuit [13]. A garbled circuit is an encrypted form of a function that allows the function to be evaluated but hides all private information about the function. In other words, an evaluation of the garbled circuit does not reveal any information about the function except the result of one evaluation.

Figure 4 illustrates the concept of a garbled circuit. In this figure, it is assumed that the function has two inputs and generates two outputs. For each input, a translation table translates a clear input to a garbled input. Then the garbled inputs are fed into the garbled implementation of the function. After the evaluation, each garbled output is translated to the corresponding clear output by using its translation table.

Take the airline-ticket agent for example. A garbled version of the update function needs two inputs: input 1, the agent’s current state; and input 2, the local input from the host, i.e., the offered flight and its price. This function generates two outputs: output 1, the agent’s updated state; and output 2, the local output to the host, i.e., the agent’s decision of whether or not to book the flight immediately.

From the construction of a garbled circuit, it is clear that the security of a garbled circuit depends on the translation tables. Figure 5 shows a sample translation table. Assume that the input or output has \( n \) bits. Then the translation table has two random strings corresponding to each bit. One of the strings will be returned.

\[ ^3 \text{If an input or output has } n \text{ bits, the size of its translation table is } O(n). \]
Figure 4: A Garbled Circuit of a Function With Two Inputs and Two Outputs

<table>
<thead>
<tr>
<th>value</th>
<th>bit 1</th>
<th>bit 2</th>
<th>...</th>
<th>bit n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td>1101</td>
<td>...</td>
<td>0110</td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>1000</td>
<td></td>
<td>1011</td>
</tr>
</tbody>
</table>

Figure 5: A Translation Table for a Data Item of \( n \) Bits
when the bit is equal to 0, while the other string is returned when the bit is equal to 1. By restricting access to the translation tables, a system can protect the privacy of a function. More specifically, the following access control must be placed on the translation tables in order to protect the function’s privacy without disabling a host’s ability to evaluate the function with its local input and get its local output.

- First, the host needs to have access to garbled inputs 1 and 2. In particular, to get garbled input 2, the host needs to be able to translate its local input to garbled input 2.

- Second, the host should be able to get the local output in clear after the evaluation. To achieve this objective, the host should have access to the translation table of output 2.

- Third, to prevent the host from knowing the state of the agent, the host should not have access to the translation tables of either input 1 or output 1.

- Fourth, to prevent the host from enumerating all possible local inputs and indirectly discovering the state of the agent, the host should be able to access the garbled input 2 corresponding to only one value of its local input. Specifically, for the update function, we need to ensure that an airline can only evaluate the outcome of one offered price; otherwise, the airline would be able to test for the student’s price threshold by gradually lowering its price.

The first three requirements are somehow easier to implement (Section 4 discusses how to meet these requirements). To implement the fourth requirement, one possibility is that the host asks for the garbled input of its local input from the originator or a trusted party, by revealing its local input. The originator or the trusted party keeps track of the query and will send only one reply. However, as discussed in Section 2.2, such a query by the host discloses the local input of the host and therefore compromises its privacy. For the general case, if the privacy of the local input of a host needs to be protected, we need to use verifiable distributed oblivious transfer.

### 3.2 Verifiable Distributed Oblivious Transfer

Verifiable distributed oblivious transfer (VDOT) is a cryptographic primitive that allows a host to receive one of two private items obliviously from some servers. Here obliviously means that the host can receive one and only one of the two items, and that the servers cannot learn which item the host receives.

Before a VDOT execution, the owner of the two private items distributes secret shares of each item among threshold-trusted servers. A host then sends a query to
each server and each server sends back a response based on its shares of the two private items. From these responses, the host is able to compute the private item it chooses, with the following security properties:

- No information about the other private item is revealed.
- The servers cannot know which item the host chooses.
- If any server cheats, the host can detect the cheater by testing an identity about the responses.

The protocol steps of VDOT are shown in Figure 6. In step 1, the host sends queries to the servers. In step 2, the servers send replies to the host, who reconstructs in step 3 the item it chooses.

![Diagram of protocol steps](image)

**Figure 6: Protocol Steps of Verifiable Distributed Oblivious Transfer**

A host can apply VDOT to retrieve the garbled input of its local input, without revealing its local input. Consider the two strings of each bit in a translation table as the two private items in VDOT (see Figure 5). We know that the host just needs to execute VDOT for each bit of its local input.\(^4\)

Consider an example in Figure 7. Assume that the translation table is that in Figure 5. For simplicity, consider only the first two bits of the local input. Assume that the values of these two input bits are 01. For the first bit, since it is equal to 0, according to the translation table, the garbled string will be 0010; for the second bit, since it is equal to 1, the garbled string will be 1000. Through two executions of VDOT, the host can retrieve 00101000, the garbled input 2 for its value of local input. For any other values of local input (i.e., 00, 10 or 11), the host cannot find the corresponding garbled input.

From the security properties of VDOT, we can derive the following security properties of a retrieval.\(^4\)

---

\(^4\)In practice, the multiple executions of VDOT can be completed in one round of communication.
The host can only obtain the entry of the table that translates the value of its local input (clear input 2) to garbled input 2. In other words, the host cannot learn how to translate any other possible value of local input, unless it corrupts more than a threshold of servers. Therefore, the host cannot evaluate the garbled circuit with any other values of local input.

The servers cannot know which entry of the table the host has retrieved. That is, the servers cannot learn about the host’s local input.

If any server cheats, the host can detect the cheater, unless more than a threshold of servers collude.

In the airline-ticket example, an airline can use VDOT to retrieve the entry that translates its offered price to garbled input 2, so that it can evaluate the garbled circuit with this offered price. The above security properties ensure that the airline cannot evaluate the garbled circuit with any other offered prices, that the servers cannot know what price is offered by this airline, and that cheater can be detected if any server cheats. It is clear that all these properties are strong security properties.

4 System Architecture and Protocol Specification

4.1 System Architecture

Using garbled circuits and VDOT allows for the creation of a system architecture to protect mobile agents under the computation model described in Section 2.1.
This architecture, shown in Figure 8, extends that in [4], using threshold-trusted servers instead of one single trusted server.

![System Architecture Diagram]

Figure 8: System Architecture

There are three types of entities in this system architecture: the originator, the hosts, and the servers. Introducing the servers allows the originator to use mobile agents without always be online. This serves a useful purpose in examples like the airline-ticket example, where the student may not have a permanent computer available. Not requiring originators to always be online also helps dial-up users, who still make up the majority of Internet users, since they do not have persistent connections. In such scenarios, the servers serve as a proxy to the originator. Note that it is straightforward to modify the system if the originator is always online.

Below we briefly discuss each of the entities.

- **Originators** The responsibility of an originator is to create an agent and send the agent to the hosts. To improve efficiency, we partition an agent into the security-sensitive portion and the general portion.

- **Hosts** The responsibility of a host is to run the general portion of an agent and interpret the garbled-circuit portion of the agent. As we discussed in the previous section, in order to interpret a garbled circuit, the host needs to run the VDOT protocol with the servers to get the appropriate entries from the translation table.

- **Servers** The responsibility of the servers is to serve as a proxy for an originator and provide translation tables to the hosts through the VDOT protocol.

### 4.2 Protocol for Security-Sensitive Computation

In order to evaluate the security-sensitive function of the agent, the originators, hosts, and servers must use the evaluation protocol described in this subsection,
which, like the architecture, extends that found in [4] by using multiple servers. The key to this protocol lies in controlling access to the translation tables of garbled circuits using VDOT.

4.2.1 Encoding of a security-sensitive function

For each host, the originator of an agent encodes the security-sensitive function by a garbled circuit and attaches the circuit to the agent. As previously discussed, a garbled circuit has four translation tables:

- (table In1) A table that translates clear input 1 (the previous state) to garbled input 1.
- (table In2) A table that translates clear input 2 (the local input) to garbled input 2.
- (table Out1) A table that translates garbled output 1 to clear output 1 (the new state);
- (table Out2) A table that translates garbled output 2 to clear output 2 (the local output).

Among the four tables, table Out2 is attached to the agent in clear text so that the host can obtain its local output immediately after the evaluation.

As for table In2, its content is split into secret shares and encrypted using the public keys of the servers. The agent carries the encrypted shares, which will be used to initialize the VDOT protocol.

<table>
<thead>
<tr>
<th>ID</th>
<th>Session identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>GbCircuit\textsubscript{j}</td>
<td>Garbled circuit for host \textsubscript{j}</td>
</tr>
<tr>
<td>GbIn1\textsubscript{Host1}</td>
<td>Garbled input 1 for host 1</td>
</tr>
<tr>
<td>GbIn1\textsubscript{Tab\textsubscript{j}(i,b)}</td>
<td>The entry of table In1 for host \textsubscript{j} when the (i)-th bit of input 1 is (b)</td>
</tr>
<tr>
<td>GbIn2\textsubscript{Tab\textsubscript{j}(i,b,m)}</td>
<td>The (m)-th share of the entry of table In2 for host \textsubscript{j} when the (i)-th bit of input 2 is (b)</td>
</tr>
<tr>
<td>GbOut1\textsubscript{Tab\textsubscript{j}(i,b)}</td>
<td>The entry of table Out1 for host \textsubscript{j} when the (i)-th bit of output 1 is (b)</td>
</tr>
<tr>
<td>GbOut2\textsubscript{Tab\textsubscript{j}}</td>
<td>The translation table Out2 for host \textsubscript{j}</td>
</tr>
</tbody>
</table>

Table 1: Notation
Tables In1 and Out1 encode the state of the agent. Note that the clear output 1 at host $j$ should be the same as the clear input 1 at host $j + 1$. As a result, their garbled versions are also correlated and there is a way to enforce the state transition without revealing these two tables. In particular, a chaining technique is used to combine the entries of table Out1 at host $j$ with the corresponding entries of table In1 at host $j + 1$, the next host [5, 4].

Figure 9 summarizes the data format carried by an agent for a security-sensitive function (the notation is explained in Table 1). In this protocol, we use both asymmetric encryption and symmetric encryption. $PE(ek, m)$ denotes the asymmetric encryption of cleartext $m$ with encryption key $ek$; $E(k, m)$ denotes the symmetric encryption of cleartext $m$ with key $k$. It is required that it be easy to verify whether or not a ciphertext is encrypted with a key in the symmetric encryption scheme. Note that this property can be implemented by adding redundancy to the cleartext before encryption.

$$
\begin{array}{|c|c|c|}
\hline
\text{ID} & \text{GbCircuit}_1 & \text{GbInput1Host1} \\
\hline
\{PE(ekm, ID)|i|m|GbInput2Tab\}_{i,b,m} & \text{GbOutput2Tab}_1 & \ldots \ldots \\
\hline
\text{GbOutput2Tab}_1 & \ldots \ldots & \text{GbCircuit}_j \\
\hline
\{E(GbOutput1Tab\ldots(i,b),GbInput1Tab\ldots(i,b))\} & \ldots \ldots \\
\hline
\{PE(ekm, ID)|j|m|GbInput2Tab\}_{j,b,m} & \text{GbOutput2Tab}_j & \ldots \ldots \\
\hline
\text{GbOutput2Tab}_j & \ldots \ldots \\
\hline
\end{array}
$$

Figure 9: Data Format of a Security-Sensitive Function in an Agent

### 4.2.2 Evaluation of a security-sensitive function

When an agent arrives at a host, since In1 is chained to Out1 of the previous host, the host uses the garbled output 1 of the previous host to retrieve its garbled input 1. The host then executes VDOT to obtain the value of garbled input 2 corresponding to its local input.

With both garbled input 1 and garbled input 2, the host evaluates the garbled circuit. After the evaluation, the host uses the attached table Out2 to get its local output.
output. Then it attaches its garbled output 1 to the agent so that the next host can retrieve its garbled input 1 from the agent.

Figure 10 shows the information flow of our protocol at a host.

Figure 10: Evaluating a Security-Sensitive Function at Host $j$

The last host sends the agent back to the originator. The originator then translates the garbled output 1 to determine the final state of the computation.

4.3 Optimizations

There are several ways to improve the efficiency of the previous protocol.

First, in the previous protocol, the originator encrypts the translation table of input 2 using asymmetric encryption ($\{PE(ekm, ID[j]|i|m|GbInput2Tab_{j}(i, b))\}_{i,b}$ in Figure 9). This can be expensive because asymmetric encryption is slower than symmetric encryption. Therefore, to improve efficiency, the originator encrypts a random key using asymmetric encryption and then encrypts the table with this random key using symmetric encryption.

Second, an agent may carry more and more data as it visits different hosts, since each host attaches some data to the agent. However, note that all the data carried
by the agent is for one-time use only, except the session ID. Therefore, before the agent leaves any host, all the data used at this host can be deleted. More exactly, $GbCircuit_j$, {$E(GbOutput1Tab_{j-1}(i, b), GbInput1Tab_j(i, b))_{i, b}$, {$PE(ekm, ID[j, j,m]GbInput2Tab_j(i, b))_{i, b}$. $GbOutput2Tab_j$ and the garbled output 1 of host j-1 can be deleted from the agent’s data when the agent leaves host j.

4.4 Security Analysis

The system just described allows for the protection of the originator’s privacy, the protection of the host’s privacy, and the detection of cheaters. These security properties derive from the security properties of garbled circuits and VDOT.

**Theorem 1** (Originator’s Privacy) The originator’s private information in the security-sensitive portion of the agent is private against any hosts and any servers, unless more than a threshold of servers collude.

This follows from the security properties of garbled circuits and VDOT. Since only the originator knows the translation tables of input 1 and output 1, the state information (in which the originator’s private information is hidden) is private against other parties. Because VDOT ensures that each host can only evaluate the garbled circuit with one value of its local input, no host is able to extract partial private information from the garbled circuit by evaluating it for more than once.

**Theorem 2** (Host’s Privacy) A host’s local input to the agent and local output from the agent are private against the originator, any other hosts and any servers, no matter how many parties involved collude.

The privacy of local input follows from the security properties of VDOT. Because the local input is not revealed in VDOT, there is no way for other parties to learn about it. The privacy of the local output is obvious.

**Theorem 3** (Cheater Detection) If any server cheats, the host is able to detect the cheater, unless more than a threshold (which is different from the threshold for the originator’s privacy) of servers collude.

This also follows from the security properties of VDOT.

5 Implementation and Performance Evaluation

While the system described in the previous section has been shown achieve the desired security properties, it would be useless if the costs of the protocol were
too high to make it feasible in practical applications. An implementation of both VDOT and garbled circuits demonstrates that the costs of using these two tools in the protocol is acceptable.

In the software design for this system, the general portion of an agent will be implemented in Java, while the security-sensitive portion will be encoded as a garbled circuit. Figure 11 shows the components and the information flow at an originator. In my current implementation, a user needs to manually generate a garbled circuit, which should be very small for many applications. In the future, an automatic circuit generator should be built, allowing a user to generate a circuit for her own use by specifying her own parameters. In the airline-ticket example, all the user needs to do is to execute the generator and input her desired flight date, source, destination and price threshold. Then the generator immediately outputs a mobile agent on her behalf.

An agent is sent to hosts for execution. Figure 12 shows the components and the information flow at a host. Since garbled circuits are general purpose and are represented in a platform-independent format, for the purpose of efficiency, our current interpreter is implemented in C.

Obviously, one potential major overhead will be the evaluation of garbled circuits. However, measurement of the prototype interpreter shows that the overhead is very small. Figure 13 shows the overhead of evaluating random garbled circuits of different sizes. The result shows that the overhead of evaluating a garbled circuit of several hundred gates is pretty small.

In order to interpret garbled circuits, the hosts need to interact with the servers to retrieve translation tables. In particular, if the local inputs of the hosts need
Figure 12: Components of a Host

Figure 13: Overhead of Evaluating a Garbled Circuit
to be protected, the hosts and the servers will interact through the VDOT protocol. Because of its robust networking and security libraries, the VDOT protocol components were implemented using Java.

5.1 VDOT Implementation Details

The VDOT implementation consists of four major components, VDOTInitializer, VDOTConfigFile, VDOTServer, VDOTSender, and VDOTReceiver. The VDOTInitializer creates the configuration file that must be used by all other components in the system. This component only needs to be run once to set up the configuration file for all subsequent VDOT transactions and must be run on a trusted server. The configuration file contains information such as the public and private keys of the servers as well as parameters needed to perform the VDOT calculations. It is used in order to simplify the necessary communication between the components of the system. A more robust implementation of this protocol would have a public key management system to control access to the public and private keys.

VDOTConfigFile is the class used to represent a configuration file created by the VDOTInitializer. The VDOTConfigFile constructor takes the name of the configuration file as a parameter.

VDOTServer is the server class that the senders and receivers will interact with in the VDOT protocol. The VDOTServer stores shares from senders and processes requests for receivers. When the server receives a share from a sender, it stores the share according to the sender's hostname. The receiver, in order to receive the appropriate share, must specify the hostname of the server it wants to receive from. Receiver requests can be either requests for shares or requests for verification of shares. The VDOTServer will send a response according to the type of request.

VDOTSender is the class used by a sender to distribute its two secrets among the VDOTServers. To use the VDOTSender class in a program, a user must first create a new VDOTSender object by calling its constructor. Next, it must set the two secret items for the VDOTSender using the setShares(String a, String b) method. This breaks up the two strings into shares to be sent to servers. Finally, the user must call the sendShares() method, which sends the shares to the servers specified in the configuration file created by VDOTInitializer.

VDOTReceiver is the class used by a receiver to send requests to the servers and reconstructs the original secret from the responses it receives. VDOTReceiver also verifies that the shares it received are correct and that none of the servers is cheating. While it can tell whether any of the shares is incorrect (veri-
fication), it does not presently implement the cheater-identification and accusation services described in [15]. To use the VDOTReceiver class in a program, the user must create a new instance of the VDOTReceiver object by calling its constructor. Then, it must call the setFrom(String hostname) method to specify the hostname corresponding to the sender it wants to receive a secret from. In order to send the requests to the servers, the receiver then calls getShares(boolean first, boolean second), where the appropriate boolean is true for the secret the receiver wants to receive. Once the receiver has received the responses, it calls combineShares() which will combine the shares and return the original secret string. The combineShares() method also verifies the shares by querying the verification servers. If a cheater is detected among the shares, the method throws an error.

5.2 VDOT Performance

We next evaluate the overhead of the VDOT protocol. The dominant operation in the protocol involves exponentiations, so the speed with which exponentiation of large numbers can be done dictates the costs of each computation. For the Java implementation, the cost of each exponentiation was around 90-100 ms, a price that is higher than desired for practical purposes. A test of the time intensive calculations recoded in C++ shows that this cost can be reduced to around 80 ms. Figure 14 shows the steps of the VDOT protocol and labels the cost of each computational step when implemented in C++. The setting of the evaluation is 6 servers and the threshold for the originator’s privacy is 3.

![Diagram of VDOT protocol](image)

Figure 14: Overhead of VDOT (Number of Servers: 6; Threshold: 3)

From a practical standpoint, the cost for companies using this protocol would lie in the ability of a company’s servers to process a workable number of requests
at any given time. For this protocol, the costs of evaluation at the server are 161.4 ms per request. This allows servers to serve approximately 6-7 requests per second (more could be processed on more powerful machines). While this is lower than probably desired, companies can use more servers if their expected capacity exceeds this number. Ultimately, the tradeoff between protecting the mobile-agent’s privacy and performance of the system would have to be decided on by each company, although future optimizations of this protocol could reduce costs and improve the system’s feasibility.

6 Conclusion and Future Work

This paper discusses the implementation of a system for mobile-agent computation that extends that found in [4] by using garbled circuits and VDOT to replace the single trusted server with a group of threshold-trusted servers. Garbled circuits allow the security-sensitive portion of the mobile agent to be evaluated without divulging the agent’s private information. VDOT enables the regulation of access to the translation tables of the garbled circuit, ensuring that the circuit can only be evaluated for one set of inputs. The combination of these two components satisfies the privacy requirements of the host and originator. The implementation and evaluation of the performance of these two key components of the system have shown that VDOT and garbled circuits can be used to protect the privacy of mobile agents and their hosts in an efficient way, since the overhead involved in their calculations is acceptable. For garbled circuits, the cost of evaluation is minimal, costing about 1 ms for every 1000 gates in the circuit. The main cost of this system lies in the VDOT protocol. While the costs here are higher than would be desired (around 160 ms for each request on the server side), they are not so high as to rule out its use. Since the cost of VDOT lies derives mostly from the cost of exponentiations of large numbers, future optimizations of this process would lower the overhead of this protocol. Ultimately, a tradeoff exists between the added security of this protocol and the processing overhead involved with its use. Those who value the added security must weigh its advantages with its costs in making a final determination of whether or not this system is useful.

7 Acknowledgements

This paper was made possible with the ideas and aid of Sheng Zhong and Yang Richard Yang of the Yale University Computer Science Department.
References


