ABSTRACT

Scaling Overlay Access Control

Lin Cai

Mentor: Michael J. Donahoo, Ph.D.

Many emerging network applications depend on services provided by overlay networks, such as, multicast, concast, distributed search, and producer/consumer anonymity. Often such overlay networks do not control access, making them vulnerable to suffering or even participating in denial-of-service attacks. Since overlays exhibit a high degree of dynamics due to interconnection optimization and member joins/leaves, we must take care to avoid communication and processing bottlenecks. We propose a scheme, called Authorization To Go (ATG), where overlay members carry credentials that allow them to locally prove their permission to overlay and service access. Credential updates are multicast via the overlay infrastructure to provide access control and forward/backward secrecy. Providing locally verifiable credentials reduces the need for a member to communicate with the authorization server, which greatly increase system scalability.
Approved by the School of Engineering and Computer Science:

__________________________
Donald L. Gaitros, Ph.D., Chairperson

Approved by the Thesis Committee:

__________________________
Michael J. Donahoo, Ph.D., Chairperson

__________________________
William B. Poucher, Ph.D.

__________________________
Corey P. Carbonara, Ph.D.

Approved by the Graduate School:

__________________________
J. Larry Lyon, Ph.D., Dean
Scaling Overlay Access Control

A Thesis Submitted to the Faculty of
Baylor University
in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science

By
Lin Cai

Waco, Texas
Aug. 2004
# TABLE OF CONTENTS

## LIST OF FIGURES

1. **Introduction** .......................................................... 1
   - 1.1 Related work .................................................. 2
   - 1.2 Our solution: ATG ........................................... 3
   - 1.3 Paper organization ............................................ 4

2. **Protocol** .................................................................. 5
   - 2.1 Overall operation ............................................... 5
     - 2.1.1 Credential .................................................. 5
     - 2.1.2 Version ..................................................... 7
     - 2.1.3 Repair loss ................................................. 8
   - 2.2 Terminology ....................................................... 9
   - 2.3 Message ............................................................ 10
   - 2.4 Protocol specification ......................................... 16
     - 2.4.1 Authorize and leave .................................... 16
     - 2.4.2 Connect to neighbors ................................. 17
     - 2.4.3 Repair ...................................................... 17
   - 2.5 Batching ........................................................... 18
   - 2.6 Algorithm .......................................................... 22
     - 2.6.1 Create a credential ................................. 22
     - 2.6.2 Verify a credential .................................... 22
     - 2.6.3 Update credential using a single credential .... 23

3. **Design and implementation** ........................................ 25
   - 3.1 Generic interface for overlay network authorization .... 25
     - 3.1.1 Interface at the client ............................... 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.2 Interface at the AS</td>
<td>26</td>
</tr>
<tr>
<td>3.2 ATG module design</td>
<td>27</td>
</tr>
<tr>
<td>3.2.1 Transaction management</td>
<td>30</td>
</tr>
<tr>
<td>3.2.2 Message</td>
<td>32</td>
</tr>
<tr>
<td>3.2.3 Timer</td>
<td>32</td>
</tr>
<tr>
<td>3.2.4 Batching control</td>
<td>32</td>
</tr>
<tr>
<td>3.2.5 Network layer</td>
<td>33</td>
</tr>
<tr>
<td>3.2.6 Configuration and log</td>
<td>34</td>
</tr>
<tr>
<td>4 Experimental results</td>
<td>39</td>
</tr>
<tr>
<td>4.1 Non-batching mode</td>
<td>39</td>
</tr>
<tr>
<td>4.2 Batching mode</td>
<td>41</td>
</tr>
<tr>
<td>5 Summary</td>
<td>43</td>
</tr>
<tr>
<td>A Signature operation comparison</td>
<td>45</td>
</tr>
</tbody>
</table>

APPENDIX

BIBLIOGRAPHY
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A sample binary hash tree</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Basic operations in ATG</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>A successful join or leaving operation</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>A successful connection set up between two nodes</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>ATG node state</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>ATG server</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>ATG node</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Transaction manager</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>ATG message</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>Timer</td>
<td>32</td>
</tr>
<tr>
<td>3.6</td>
<td>Network</td>
<td>33</td>
</tr>
</tbody>
</table>
CHAPTER ONE

Introduction

An overlay network is a set of self-organizing entities connected with virtual point-to-point communication channels. To maintain and/or improve performance, overlay nodes may frequently change their interconnections. These entities provide a flexible infrastructure to implement a wide variety of network services, such as content-based routing (Ratnasamy, Francis, Handley, Karp, and Shenker 2001), producer/consumer anonymity (Clarke, Sandberg, Wiley, and Hong 2000), and group communication (Chu, Rao, and Zhang 2000) (Francis, Pryadkin, Radoslavov, and Govindan 2002). Although some of these services can be integrated in network layer, the difficulty in deployment and constrained memory and computation in network layer make this solution difficult.

In practice, most overlay networks ignore the issue of access control, which means any node can connect to overlay nodes. This may cause several problems. First, the services provided by overlay networks can be comprised by malicious nodes. For example, in BitTorrent (Cohen 2003), a modified node can choke all peers while still downloading from other nodes. Second, such nodes can disturb or even destroy the topological structure of the overlay. In CAN (Ratnasamy, Francis, Handley, Karp, and Shenker 2001) or CHORD (Stoica, Morris, Karger, Kaashoek, and Balakrishnan 2001), a node can easily cause a routing loop by changing its routing table. Third, uninvited nodes increase the cost for valid overlay nodes. For example, in a video stream broadcast overlay, adding a new subscriber means an existing node has to forward an extra copy of the stream. Finally, overlay networks may provide hosts to attack outside services. As mentioned in (Zeinalipour-Yazti 2002), a malicious node in Gnutella can launch a Distributed Denial-of-Service (DDoS) attack by routing every query to a single target node.
Current solutions to these problems either exhibit limited scalability or focus on securing data. We propose a novel, scalable authorization method that enforces the past and future secrecy by assigning each member a credential. With modest cost of memory and communication, every member is able to verify other’s credential locally and set up connections freely without the help from a central server.

1.1 Related work

In this section, we present work related to overlay access authorization. We consider two basic approaches: Access Control List (ACL) and data encryption.

The first approach uses some form of ACL, which includes the identities of valid nodes. In the Core Based Tree (CBT) model (Balardie, Crowcroft, and Francis 1993), the initiator of a group creates an ACL and sends it to the primary core of the group together with its digital signature. The primary core then creates a group key and a key to encrypt future keys (KEK). The ACL, group key and KEK are disseminated to secondary cores. All nodes in cores make up a distribution tree, and any of them can authenticate any member in the ACL. Before transmission begins, a session key is distributed to the authenticated nodes. Nodes not listed in the ACL are not able to join the group or decrypt data. The distribution of a single ACL is not feasible for a large group and does not take care of member changes. To improve the scalability, in Iolus (Mittra 1997), the group list is divided into several subgroups. One Group Security Controller (GSC) manages the top-level subgroup. Each subgroup is then controlled by a Group Security Intermediaries (GSI). A node sends join requests and receives group keys through a secure unicast to a GSI. This GSI check the node’s membership in a local ACL. Each node must send refresh messages to the GSI to prevent membership expiration. If the group size is n+1, when a member leaves, the GSI either sends out the updated group key in n unicast messages or multicasts one message that includes n copies of the group key individually encrypted by one
remaining member’s key. Consequently, the leave operation is still expensive in Iolus.

The second approach depends on encrypting group data with a secret key. Any
node can join the group and receive data; however only members know the secret key
necessary for data decryption. Obviously, the group key must be distributed securely
within a group whenever the group nodes change. After a node joins, it is easy for
the server to distribute the new group key encrypted by the old group key. The
difficult part is to distribute the new group key after a node leaves. One approach
is to divide all nodes in the group into many subgroups that may be overlap with
each other. Each subgroup maintains its own key. When a node leaves, the server
can send the new group key to nodes that are not in the same group as the leaving
node and encrypt the new key with those subgroup keys. One approach to forming
subgroups involves constructing a tree where the leaves represent members, the root
stores the group key, and internal nodes holds the subgroup keys (Wong, Gouda, and
Lam 1998) (McGrew and Sherman 1998). These methods are scalable for maintaining
the group key, but this approach only limits data access, not overlay membership or
data forwarding.

1.2 Our solution: ATG

We propose a novel solution called ATG (Authorization To Go) for overlay
access control and data forwarding to provide the past and future secrecy, which
means a node is not allowed to read the overlay data before its join or after its leave.
Our approach increases scalability by allowing nodes in the overlay to connect with
each other using local exchange for verification. There are two components in our
solution: the Authorization Server (AS) and the node. Each node and the AS are
identified by a public/private key pair. The public key of the server is well-known.
The AS manages all operations of authorization and leaving.

When a node wants to join a group, it sends the join request to the AS. The
AS will verify the node and create a credential if the node is valid. This credential, including a time stamp of join, is signed by the AS. After receiving and verifying the credential, the node knows the result of the join request. After a node joins the overlay, we call it a member. Each member maintains its own credential via updates broadcast on the overlay. A member receiving a connection request can verify the credential locally, if the member keeps its own credential fresh according to the update broadcasts. Because every credential includes a time stamp as well as group data, a member is able to make forwarding decisions with respect to its neighbor’s time stamp, which ensures the past and future secrecy. Data is only forwarded to members in the group to avoid communication overhead. When a member leaves the group, the AS also generates a credential and broadcasts it in the overlay. Therefore, in ATG, the central AS is only responsible for authorization and credential maintenance. Overlay access control and data forward decisions are distributed among each member to improve scalability. ATG can be with any overlay providing best effort unicast and broadcast.

1.3 Paper organization

The paper is organized as follows. In Chapter 2, we discuss details about the ATG protocol. Chapter 3 explains some important issues on our implementation. Chapter 4 details our experiments and analysis of the result data. The final chapter summaries our work and proposes some future directions in this area.
CHAPTER TWO

Protocol

The ATG (Authorization To Go) protocol provides overlay access control by allowing overlay members to decide if a requesting node may connect. In addition, it directs overlay forwarding decisions to provide past and future secrecy. This specification defines the protocol of version 1.0.

2.1 Overall operation

An ATG Authorization Server (AS) provides membership management for an overlay. When a node decides to join an overlay, it first sends a join request to the AS. The AS challenges the node to make sure this is an authenticate request. Then the AS provides a credential. With the credential, the node connects to members within the overlay using only local communication. Credentials can later be revoked by the AS to evict nodes from the overlay.

2.1.1 Credential

In ATG, a member’s credential is the proof of its membership. Any member with fresh credential can verify the correctness of another member’s credential locally without the help from the AS. Such credentials are analogous to a passport, which enables a person to prove his identity to local authorities without calling the government office. The naive approach implementing credentials is to list the identity of all overlay members in an access list. The AS must sign this list to prevent forgery. If a node can prove its identity is in this list, it will be admitted by some member of the overlay. Obviously, this method has limited scalability, because the size of the credential is proportional to the size of the overlay membership.

We build a binary tree similar to the idea in (Wong and Lam 1999) for cre-
Figure 2.1. A sample binary hash tree

dentals. Every member is placed at a leaf, and the value of the leaf node is the cryptographic hash of the corresponding member’s important information, which may include the public key, the join time, and some application specific data. The value of internal node is the hash of the concatenated values from its two children. For example, in Figure 2.1, R1 is a member in the overlay. The value of node 2.1 is the hash value of R1’s important information. The value of node 1.1 equals the hash of the values of 2.1 and 2.2. We can see the root hash value covers all nodes in the tree. Every leaf has a unique path to the root node, which is called its hash path. Every tree node in the hash path has its sibling node, except the root node. These sibling tree nodes compose the sibling path. Given a node Ri and its sibling path, we can calculate the root hash. In Figure 2.1, R1’s sibling path contains the value of 2.2, and 1.2. Using 2.1 and 2.2, we get the value of 1.1. Next, we concatenate the values of 1.2 to the 1.1 and compute the root hash. The signed root hash, the sibling path,
and the member’s hash value is the primary data structure of our credential. After a node joins the overlay, it will receive its credential, which proves it is a member in this overlay. To access the overlay, a node sends its important information and credential to a member. After verifying the authenticity of the important information, the member computes the root hash. If the result matches the root hash sent from the AS, this node is admitted as a member.

Any change in membership causes the root hash to change as well. When a member leaves, its leaf node in the hash tree is reset to blank. A credential is also issued, but this credential has no value to the leaving node. All these credentials with a signed root hash are broadcast to the whole group to help other members update their own credentials. When a member receives a credential broadcast, it first computes a root hash out of the credential and compares it with the signed root hash to verify this broadcast. Then it will compare its sibling path with the sibling path received in the credential and update any affected node in its credential. The goal is to verify that its own credential can compute the same root hash as the new one. The end of this chapter gives details about several important algorithms for using credentials.

2.1.2 Version

Whenever a node, n, joins or leaves, the hash tree nodes along the path, from n to the root, are changed. Each new root hash is given a sequentially increasing version number, which may help a member to detect the loss of some credential broadcast. A root hash with its version is called a root token. The server signs the root token to prove its authenticity, which prevents malicious node from forging it. In order to protect the past and future secrecy of the group, ATG requires every data packet to carry a version number. When a member initiates a packet, it attaches the current verified version, x, received from the AS to the data packet. Then it checks every
current valid neighbor $i$ with join version $K_i$. If $x$ is less than $K_i$, it will not send this packet to this neighbor. When any neighbor receives the packet, it does the same checking. Therefore, invalid nodes are not able to see any data which is not allowed to see, if every member obeys this forwarding rule.

2.1.3 Repair loss

A member in the overlay holds a current version and expects the credential broadcast with the next version, which is the current version plus one. Reception of a valid broadcast with some bigger version indicates the loss of some credential broadcast causing the member to set itself in the stale state as showed in Figure 2.5. A stale node may recover its fresh credential with the help of other nodes or get it directly from the AS. In Figure 2.2, Node A, B, C communicates with the AS at time 0, 1, 2 individually to join the overlay. Node A loses the broadcast at time 1, it asks AS to repair its credential. When Node C leaves at time 4, Node B does not receive this broadcast. It asks Node A to give it the broadcast credential at time 4.

To minimize the traffic to the AS, the stale node always tries to get missing
credentials from its neighbors first. If this fails, the node requests help from the AS finally, and the server sends the node’s current credential instead of those missing credentials.

2.2 Terminology

In this section, we give brief definition about important concepts in ATG, although some of them may have been mentioned before:

Version A long integer that is incremented by 1 after each join or leave

Member Hash A hash value of a node’s public key, join version, and some important data represented as a byte array. The server must guarantee every member hash is different.

Hash Tree A binary tree in which a leaf node has a member hash of one node in the overlay group and an internal node’s value is the hash value of its two children’s hash values concatenated from the left to the right

Root Hash The value of the root node in the hash tree

Root Token The root hash plus the current version

Credential Record A node in the hash tree represented as a byte indicating left or right child and a byte array which is the hash value of the node

Hash Path The set of credential record from the root to the leaf of a node. In Figure 2.1, the hash path for R1 is (0, 1.1, 2.1).

Sibling Path The set of nodes that are the corresponding sibling of each node in the hash path. In Figure 2.1, the sibling path for R1 is (1.2, 2.2).

Credential The sibling path in reverse order that is a set of credential records from its leaf node to the root. Every node has its own credential. In Figure 2.1,
the credential for R1 is (2.2, 1.2). Every credential also includes a root token to indicate its version.

**Stale** A node is stale if its credential’s version is less than the version of the hash tree.

### 2.3 Message

In this section, we give the format of messages in ATG as the following:

Data types used in ATG message

- **integer** Four octet integer encoded as two’s complement
- **long** Eight octet integer encoded as two’s complement
- **byte array** A prefix-length encoded byte array

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LENGTH</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LENGTH** An integer to indicate the length of the following byte array

**DATA** A byte array contains the payload.

**PACKET** The general format of packet in ATG protocol. Every ATG message is conveyed inside Message field of a PACKET.

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VERSION</td>
<td>RESERVED FIELD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TYPE</td>
<td>MESSAGE BODY</td>
<td></td>
</tr>
</tbody>
</table>

**VERSION** The version of ATG protocol (current version: 0x01)
RESERVED FIELD  Three octets reserved field (not used in the current version)

TYPE  One octet containing one of the following type codes:

MSG_AUTH_REQ  10 an authorization request
MSG_AUTH_ACK  11 an authorization acknowledge
MSG_AUTH_NAK  12 an authorization failure
MSG_LEAVE_REQ  20 a leave request
MSG_LEAVE_ACK  21 a leave acknowledge
MSG_LEAVE_NAK  22 a leave failure
MSG_BC_AUTH   51 a broadcast of join
MSG_BC_LEAVE   52 a broadcast of leave
MSG_CHALL     40 a challenge
MSG_CHALL_RSP 41 a challenge response
MSG_CONN_REQ   30 a connection request
MSG_CONN_ACK   31 a connection acknowledge
MSG_CONN_NAK   32 a connection failure
MSG_REPAIR_REQ 60 a repair request
MSG_BC_BATCHING 70 a batched broadcast

MESSAGE BODY  A variable length octets for the body of a message. Its format varies according to the TYPE.

PUBK  A public key encoded in X.509 format(Group 2004) stored as a byte array

RTOK  A root token
ROOT HASH  A root hash encoded as a byte array

VERSION  A long indicating the version of the root hash

SIGNATURE  A byte array storing the signature of the previous ROOT HASH and VERSION. The payload is encoded in PKCS #8 format (RSA Data Security 2004)

CREREC  One node in the hash tree

INDEX  One octet indicating a left child (0x01) or a right child (0x02)

NODE  A node's hash value encoded in a byte array

CRED  A credential for a node including a set of credential records

ROOT TOKEN  A root token encoded in a RTOK, which can be verified by the following credential record using the algorithm described in 2.6.2

CREDENTIAL RECORD  Zero or many credential records encoded in CREREC
REQ A general request message

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC KEY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFORMATION</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PUBLIC KEY The requester’s public key encoded in a PUBK

INFORMATION A byte array storing some application specific information

Message body of MSG.AUTH.REQ An authorization request

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CREMSG A general message contains credential. The meaning of each field is specified in individual message.

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC KEY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VERSION</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFORMATION</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREDENTIAL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PUBLIC KEY The public key of this credential’s owner encoded in a PUBK

VERSION A long to indicate the credential’s version

INFORMATION A byte array describing some application specific data about the owner

CREDENTIAL A credential of the owner encoded in a CRE

Message body of MSG.AUTH.ACK An authorization response from the server

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREMSG</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In CREMSG:

**PUBLIC KEY**  The authorized node’s public key

**VERSION**  The time version of this authorization

**INFORMATION**  Some application information about the node

**CREDENTIAL**  The node’s credential

**Message body of MSG.LEAVE.REQ**  A leave request sent from a node to the server

```
0  8  16  24

REQ
```

**Message body of MSG.LEAVE.ACK**  A leaving approval from the server

```
0  8  16  24

CREMSG
```

In CREMSG:

**PUBLIC KEY**  The left node’s public key

**VERSION**  The time version when the node was left

**INFORMATION**  Some application information about the node

**CREDENTIAL**  The credential

**Message body of MSG.AUTH.NAK**  Empty. Indicates a fail authorization.

**Message body of MSG.LEAVE.NAK**  Empty. Indicates a fail leaving.

**Message body of MSGCONN.NAK**  Empty. Indicates a fail connection.

**Message body of MSG.BC_AUTH**  A broadcast message for authorization. It has the same format as MSG.AUTH.ACK except message type equals 51.

```
0   8  16  24

MSG.AUTH.ACK
```
Message body of MSG_BC_LEAVE A broadcast message for leaving. It has the same format as MSG_LEAVE_ACK except message type equals 52.

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MSG_LEAVE_ACK</td>
<td></td>
</tr>
</tbody>
</table>

Message body of MSG_CHALL A challenge request

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CHALLENGE</td>
<td></td>
</tr>
</tbody>
</table>

CHALLENGE The challenge bytes encoded in a byte array

Message body of MSG_CHALL_RSP A challenge response

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RESPONSE</td>
<td></td>
</tr>
</tbody>
</table>

RESPONSE An encrypted challenge by the receiver using RSA(RSA Data Security 1991) stored in a byte array

Message body of MSG_CONN_REQ A connection request to a neighbor

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CREMSG</td>
<td></td>
</tr>
</tbody>
</table>

In CREMSG:

PUBLIC KEY The sender’s public key

VERSION The time version when the sender was authorized

INFORMATION Some application information about the sender

CREDENTIAL The sender’s credential

Message body of MSG_CONN_ACK Empty. Indicates a connection request approved
Message body of MSG_REPAIR_REQ A request for fresh credentials

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PUBLIC KEY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VERSION BEGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VERSION END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PUBLIC KEY The public key of the sender encoded as a PUBK

VERSION BEGIN A long indicating the begin version requested

VERSION END A long indicating the ending version requested

2.4 Protocol specification

This section defines the message protocol for the various scenarios in ATG protocol. Any peer of the communication must send out the specific message according to the flow. If it receives any unexpected message, it may send out an abort message or not send anything to wait for the correct message until the timer expires.

2.4.1 Authorize and leave

After a node decides to join an overlay, it first sends MSG_AUTH_REQ to the ATG server. If this node has not been authorized before, the server sends a MSG_CHALL_REQ to it. Once authorized, the server calculates its current credential and directly sends it a MSG_AUTH_ACK.

The server also broadcasts this join to all nodes in the overlay in a MSG_BC_AUTH. The message flow of a typical successful join operation is shown in Figure 2.3. Any error during this operation causes the server or the node to send a MSG_AUTH_NAK to abort the whole process. The leave operation is identical to the join except message types are different.
2.4.2 Connect to neighbors

An authorized node in an overlay group can connect to any other node in the same group by verifying each other’s credential. In Figure 2.4, Node A wants to connect with Node B. Node A sends a MSGCONN_REQ to Node B. Node B verifies A’s credential in this message and sends its own credential if A’s credential is valid. Then they both challenge each other with some random bytes. After challenge and response, Node A and B send MSGCONN_ACK to confirm the connection. If either A or B detects an error, it can send MSGCONN_NAK to abort the connection.

2.4.3 Repair

A node checks every version received in any packets. If the version is greater than what it is expecting, this node marks itself as stale and places the sender of this future version in a repair candidate list. If the time that a node is in the stale state is
Figure 2.4. A successful connection set up between two nodes

longer than a preset threshold, it chooses a candidate to send a MSG_REPAIR_REQ, which contains the range of missing versions. The receiver of the repair request, either a node or the server, attempts to update this stale node. If it is a node which happens to have some of the requested credentials, it sends those credentials back in MSG_BC_AUTH or MSG_BC_LEAVE format. If it is the AS, it computes the current credential for the requester and also sends out a MSG_AUTH_ACK. Whenever the stale node collects all missing credentials or receives a fresh one from the AS, the node goes back to Member state, as showed in Figure 2.5

2.5 Batching

In every join and leave, the server issues one signed root token for each transaction. Unfortunately, the signature operation is computationally expensive, limiting the scalability of the server. To reduce the total signature operations, we allow
Figure 2.5. ATG node state
the server to batch several transactions together into a single operation requiring one signature. In batching, the version number only changes once for all transactions with a batch. The broadcast message MSG_BC_AUTH or MSG_BC_LEAVE is not enough for this batching operation. The nodes in the group need more details about the hash tree to update their own credentials. This is the reason we introduce MSG_BC_BATCHING. In the batching mode, an ATG server delays the MSG_AUTH_ACK and the broadcast until some threshold. Then the server broadcast a MSG_BC_BATCHING instead of MSG_BC_AUTH, see Figure 2.3, which includes multiple joins and leavings accumulated so far. The batching method decreases substantially the computation of signature on the server side. In MSG_BC_BATCHING message, the hash tree is described in a post-order traverse. Each node is marked as an internal node, a leaf node or a node unchanged. Therefore, the unchanged parts of the tree are pruned from the description to decrease the message size. When a node receives such a description, it can easily update the hash values interested. For example, in Figure 2.1, if R2 leaves and a new node R4 is put into R2's place, the server sends out a batched message like this:

leaf node 2.1

leaf node 2.2

internal node 1.1

unchanged node 1.2

internal node the root

Node R1 updates its credential only for 2.2 node and skip 1.2. If there was a big tree under node 1.2, the message size would be decreased significantly. We give the format of message MSG_BC_BATCHING as the following:
**TNODE** A node in the Hash Tree sharing the same format as credential record but having different meaning on the index field.

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREREC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In CREREC

**INDEX** One octet to indicate an internal node (0x0A), leaf (0x14), or unchanged node (0x63)

**NODE** The hash value of a hash tree node

**Message body of MSG_BC_BATCHING** A broadcast message of batched updates.

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROOT TOKEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREE NODE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ROOT TOKEN** The root token of the hash tree encoded in a RTOK

**TREE NODE** A traverse of the hash tree in post-order including one or many TNODE’s

The next question about batching is to control how many transactions to queue up for the server in one batching. If the server queues too few transactions, the performance improvement is not obvious. If the server is too greedy, the client with pending transaction may time out. We use two thresholds for batching. One is the size of the batching queue. The server fires a batching when the queue reaches this size. Another is a timer. If it expires, the server also does a batching even if the queue is not full. This timer guarantees the client does not time out even if the server is starving.
2.6 Algorithm

In this section, we give important algorithms about credential operations including its creation, verification and update.

2.6.1 Create a credential

The server creates a credential for a node in the Hash Tree, which is a set of nodes in the tree ordered from the leaf of that node to the root.

```java
// Algorithm to create a credential
// Return a Credential
//
Credential createCredential(Member a, // a member to create the credential for }
{ TreeNode node=a.getTreeNode(); // get the tree node of this member Credential rtn=new Credential(); // create an empty credential while(node is not the root){
   parent=node.parent(); // get the parent node
   // get the sibling node, create a record and append to the credential
   if(node is the left child)
      rtn.append(new CredentialRecord(left, parent.right.hashvalue()));
   else
      rtn.append(new CredentialRecord(right, parent.left.hashvalue()));
   node=parent;
} return rtn; }
```

2.6.2 Verify a credential

Whenever a node receives a credential from other node, it has to verify the credential using the sender’s public key, join version and its extra information bytes.
The following is the algorithm to verify a credential.

```
//
// Algorithm to verify a Credential
// Return true if the Credential is valid
//
boolean verifyCredential( public key pk, //the public key of the sender version ver, //the join version of the sender bytes data, //the extra information Credential c //the credential to verify ){
write pk, ver, data into a byte array B; h=hash(B);
it=c.getNextCredentialRecord(); while(it is not null){
    if(isLeftChild(it))
        h=it*h; //it is a left child
    else
        h=h+it; //it is a right child
    h=hash(h);
    it=c.getNextCredentialRecord();
} if(h==c.getRootHash())
    return true;
else
    return false;
}
```

2.6.3 Update credential using a single credential

After verification of a credential, a node can use it to update the current credential if the version of it is the one expecting.

```
//
// Algorithm to update my Credential
```
boolean updateCredential( Credential my, Credential rec ){
    // a credential is a path bottom-up, reverse it to get a top-down path
    reverse(my); reverse(rec);
    // in the same sub tree
    itMy=my.getCredentialRecord()
    itRec=rec.getCredentialRecord()
    while(itMy is not null){
        if(itMy.getIndex() == itRec.getIndex())
            replace itMy hash value with itRec hash value;
        else
            break;// not my interests any more
            itMy=my.getCredentialRecord();
            itRec=rec.getCredentialRecord();
    }
    // tree splits, the old node is push to the left child by default
    if(itRec is not null)
        if(itRec is a right child)
            itMy.insert(itRec);
        else
            return false;// the new member must be in the right
    // not allowed to insert more than one member in non-batching mode
    if(itRec.getCredentialRecord() is not null)
        return false;
    return true;
CHAPTER THREE
Design and implementation

We implemented the ATG service library, a simple AS and client in Java. In this chapter, we begin by introducing the generic interface for overlay network authorization. Next, we describe the design of ATG module.

3.1 Generic interface for overlay network authorization

The authorization layer of an overlay network provides all functionalities related to membership control and secure data communication. We construct a generic set of interfaces for authorization, which are independent of the underlying authorization algorithm. To avoid blocking behavior for authorization requests, our interface uses a callback mechanism to asynchronously return authorization request results. Additionally, the node may want to cancel some request before the authorization layer returns the answer; therefore, our authorization layer includes a mechanism to handle cancellation. The following sections describe our generic authorization interface for the server and client sides individually.

3.1.1 Interface at the client

A node in the overlay network implements the client side of the authorization layer. Besides join and leave, the node asks permission from the authorization layer before it starts sending forwarding packets or connecting to other nodes. In some overlay networks, every packet is sent with a sequence number. The authorization layer is responsible for giving a sequence number to a new packet, checking the sequence number for packets received, and authorizing the forwarding of these packets. When a node, Ni, wants to connect to another node, Nj, it must verify the membership of this node. Ni submits a request to the authorization layer to check the membership
of Nj. Nj's authorization layer also will be called implicitly when receiving such a request. We summarize these requirements and provide the following functions in the interface for the client:

**abort** The client of authorization layer call this function to abort any pending function.

**authForwarding** Authorize a forwarding of data to a node: the authorized data is passed back to the caller in the callback interface.

**authMyself** The client asks the authorization layer to get the membership for an overlay group.

**authNodeActively** Initialize a verification of another node

**authNodePassively** Response to a verification request from other nodes

**getCurrentToken** Ask the authorization layer for a sequence number to append to a data packet

**leave** Leave the overlay

**removeNode** Remove a node and tell the authorization layer not to handle any request from this node

3.1.2 **Interface at the AS**

Membership control may be managed through an AS. Typically, the AS is independent from the overlay nodes. The server side authorization layer does not provide any service to the overlay layer, but it may need some service provided by the overlay network, such as broadcast to distribute credential updates. The semantics for the membership of an overlay group is only known by the specific application. When the AS is making a join decision, it has to let the application layer analyze
the semantics of the node’s information, which means the authorization layer needs the service from the application layer. Also, the application can arbitrarily tell the authorization layer to remove a node. These interfaces are simply described as the following:

**Interface of authorization layer** The application layer uses this interface to remove a node.

**Interface of overlay layer** The authorization layer uses the broadcast function in this interface, which is implemented by the overlay layer.

**Interface of application layer** The authorization layer calls functions in this interface to check a node’s application information.

### 3.2 ATG module design

The system is divided into four primary modules at both the AS and the node sides:

**Network Layer** Provide network communication to other modules

**Overlay Layer** Construct overlay topology and provide group communication to other modules

**Authorization Layer** Provide overlay network security

**Application Layer** Provide a specific application using overlay services

The relationship of these modules is shown in Figure 3.1 for the server and Figure 3.2 for the node. There are also some supporting modules for these main modules:

**Control Module** Configure, start or stop all modules

**Timer Manager** Provide timers for other modules
The server won't receive any broadcast message.

Figure 3.1. ATG server
The authorization client won't send any broadcast message.

Figure 3.2. ATG node
3.2.1 Transaction management

Each authorization or leave operation has several messages in the exchange between the node and the server, which can be grouped as one transaction. Any error occurring during the transaction aborts the process. In ATG module, there is a transaction manager handling these transactions. It creates transactions for new connections or demultiplexes messages to the proper existing transaction. A transaction is responsible for removing itself from the manager when it finishes its job or something goes wrong. A transaction also has the ability to detect the reception of incorrect messages. It can then choose to ignore the error or abort the transaction.
Figure 3.4. ATG message
3.2.2 Message

The user can implement a receiver for a specific message format and hook it with a channel. The receiver for the ATG protocol parses ATG messages from the channel and delivers to the transaction manager. The transaction manager first checks for specific the ATG message type and then handles the demultiplexing. For example, if it is a MSG_AUTH_REQ, a new authorization transaction is created and the following messages from the same channel are delivered to the new transaction. If a MSG_CHALL_REQ is received and there is no transaction from the same channel, the message will be dropped. All ATG message classes are inherited from ATGMessage, which has the toBytes() function to convert a message to a byte array. The relation between messages are illustrated in Figure 3.4.

3.2.3 Timer

If steps in a transaction have time constraints, the transaction can register a timer with a callback function. The timer is placed into the queue of the timer manager. When the timer expires, the manager calls the specific callback.

3.2.4 Batching control

There are two thresholds to control when the server flushes the batched transactions. One is the number of the transactions, which is decided by the packet size
of the credential broadcast, to avoid sending out extremely large credential update packets. Another is the maximum time for batching. When either of them exceeds the threshold, the server processes all batched transactions.

3.2.5 Network layer

The network layer provides network communication service to other layers. When a packet comes in from the network, it creates a new channel between the receiver and sender, specified by the client of the network service. After a channel receives a complete packet, it notifies the channel receiver. When some other layer wants to send a packet, it uses some existing channel or asks the network layer to create a new channel.
3.2.6 *Configuration and log*

Configuration is an interface to a name/value pair data repository with get-Value() and setValue() functions. Given a Configuration, a module can query its parameters using this interface. In the ATG module, a class called ATGConfiguration implements Configuration and reads all values from a file. ATGServerConfig and ATGClientConfig inherit ATGConfiguration to do some specific configurations at the client and the server side individually. The format of profile is very simple. Each line of the profile is of the form *parameter = value* (e.g., PORT=46411). The following provides details about all parameters:

**PORT** the listening port for ATG layer

- **data type** integer
- **constraints** any unused port number
- **example** 46411

**KEY_FILE** the file name for the node’s key pair

- **data type** string
- **constraints** none
- **example** key.txt

**SIGNATURE** true if the ATG layer actually does the signature computation

- **data type** boolean
- **constraints** none
- **example** true

**LOG_LVL** the log level
**data type** enumerated type \{ALL, WARNING, INFO, FINE, FINER, FINEST, OFF\}

**constraints** none

**example** ALL

**LOG_FILE** the name of the log file

**data type** string

**constraints** none

**example** log.txt

**LOG_NAME** the log name

**data type** string

**constraints** none

**example** edu.baylor.cs.cai.atg.server

**KEY_ALGO** the key pair algorithm

**data type** string

**constraints** must be specified by the key provider

**example** DSA

**KEY_PROVIDER** the provider of key pair algorithm

**data type** string

**constraints** the name as mentioned in the provider’s documentation

**example** SUN

**KEY_SIZE** the size of the key
**data type** integer

**constraints** must be valid for the algorithm

**example** 1024

**HASH_ALGO** the message digest algorithm

**data type** string

**constraints** must be provided by the key provider

**example** SHA-1

**CHALL_LEN** the length of the challenge bytes

**data type** integer

**constraints** positive number

**example** 4

**SIGN_ALGO** the signature algorithm

**data type** string

**constraints** must be supported by the provider

**example** SHA1withDSA

**SHORT_TIMER** a short timer

**data type** integer

**constraints** positive number

**example** 1000

**NORMAL_TIMER** a normal timer

**data type** integer
**constraints** positive number

**example** 3000

**LONG_TIMER** a long timer

**data type** integer

**constraints** positive number

**example** 10000

**BATCHING** true if the server is in batching mode

**data type** boolean

**constraints** none

**example** true

**BATCHING_TIMER** the timer for flushing in batching mode

**data type** integer

**constraints** positive number

**example** 100000

**BATCHING_SIZE** the maximum number of batched requests

**data type** integer

**constraints** positive number

**example** 100

**SERVER_KEY** the server’s public key file on a client

**data type** string

**constraints** on client side
example server.dat

SERVER.IP the server’s IP

data type IP address

constraints no

e xample 129.62.148.17

SERVER.PORT the server’s listening port

data type integer

constraints any unused port number

e xample 46411

The ATG module has a log with five levels importance to record different events:

ALL To record all events

WARNING To record error messages

INFO To record module level events, such as module start and stop

FINE To record transaction level data and events for testing data analysis

FINER To record details events for debugging

FINEST To record some temporary events for debugging

OFF Not to record any event
CHAPTER FOUR

Experimental results

This chapter details our evaluation of ATG’s scalability. In ATG, computation at the nodes naturally scales. Conversely, the AS serves for all join and leave requests, so it is more likely to become the bottleneck. To analyze the scalability of the AS, we measure the average transaction time and message size at the AS under different overlay sizes and client request frequencies. The degree to which both increase as the overlay grow indicates the scalability of our approach.

We implemented a prototype authorization server and a multiple client simulator in Java 1.4.2. The server broadcasts credential updates in UDP packets. The client simulator sends join or leave message to the server via UDP, and it can simulate one or more clients concurrently. We ran the experiments on two identical system with Intel Pentium(R) 4 1.8GHz CPU, 512MB RAM and Linux kernel 2.4.20. The client and the server are connected by 100MB local network.

4.1 Non-batching mode

We begin with a non-batching server where each request is processed separately. The AS was preloaded with an initial client population, which means the ATG server inserts a number of members into the hash tree when it starts. This internal loading time is not included in our measurement. All identities in this preloaded population are known a priori by both the server and client simulator. When the server is ready, the client simulator randomly selects a node inside this population to do a leave transaction followed by a join. Then the simulator picks the next node randomly until the total number of join and leave transactions reaches 10,000. To measure how fast the AS can process a single transaction, all these transactions are issued serially from the client simulator. The transaction time is measured at the server side. This
Table 4.1. Non-batching server transaction time and message size

<table>
<thead>
<tr>
<th>Initial population</th>
<th>Average Time (ms)</th>
<th>Average size (byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>75</td>
<td>619</td>
</tr>
<tr>
<td>3,000</td>
<td>73</td>
<td>661</td>
</tr>
<tr>
<td>10,000</td>
<td>76</td>
<td>703</td>
</tr>
<tr>
<td>100,000</td>
<td>76</td>
<td>787</td>
</tr>
<tr>
<td>300,000</td>
<td>78</td>
<td>826</td>
</tr>
<tr>
<td>1,000,000</td>
<td>92</td>
<td>868</td>
</tr>
</tbody>
</table>

ignores network latency into our measurement, which is not central to our ATG algorithm. The server records the time of the first message received as the beginning of the transaction and the time of the last message it sent out as the finishing time. The server also records how many bytes are sent out by itself in each transaction. The total time and message size sent out by the server for 10,000 transactions are recorded to calculate the average time and message size.

Table 4.1 shows the results of our experiment. When the overlay membership size increases from 1,000 to $10^6$, the average transaction time grows from 75 ms to 92 ms, and the average number of bytes sent out by the AS per transaction grows from 619 bytes to 868 bytes. Note that the time and message size grow less than 20% while the overlay size grows 1,000 times larger.

The reason that transaction time and message size grow only logarithmically with respect to overlay size is most computation time is dominated by the signature operation. When the AS receives a join or leave request, it updates the hash tree and calculates the current root hash. Then the server signs this root hash and sends out a new credential. The hash tree is about 10 levels deep for a population of 1,000 and 20 levels for $10^6$. The server does only slightly more work to insert or delete a member in a $10^6$ node hash tree when compared to the 1,000 node hash tree. The overhead of increased overlay size is almost negligible compared to the expense of the signature operation.
Table 4.2. Batching server transaction time and message

<table>
<thead>
<tr>
<th>100 concurrent transactions</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Batching Size</td>
<td>Average Time(ms)</td>
<td>Throughput</td>
</tr>
<tr>
<td>1</td>
<td>1816</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>236</td>
<td>28</td>
</tr>
<tr>
<td>30</td>
<td>538</td>
<td>35</td>
</tr>
<tr>
<td>50</td>
<td>1058</td>
<td>29</td>
</tr>
<tr>
<td>100</td>
<td>1952</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>300 concurrent transactions</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>531</td>
<td>72</td>
</tr>
<tr>
<td>30</td>
<td>436</td>
<td>75</td>
</tr>
<tr>
<td>50</td>
<td>627</td>
<td>76</td>
</tr>
<tr>
<td>100</td>
<td>1022</td>
<td>77</td>
</tr>
</tbody>
</table>

4.2 Batching mode

From the previous experiment, we now know the most expensive job for an ATG server is the signature operation; therefore, we expect better performance in the batching mode. The server only creates one signed root token when the number of queued transactions reaches the batching size or the batching timer expires. For this experiment, we need clients that are able to quickly send requests to fill up the batching queue, so in this experiment, multiple clients are simulated. To better reveal the performance of the AS handling multiple requests, we also measure its throughput, in transactions per second. It is very easy to calculate the throughput in the previous experiment. Because transactions are processed sequentially, the throughput equals 1,000 divided by the transaction time in millisecond. For example, if the transaction time is 75 ms, then the throughput is about 13.

In this experiment, the server was preloaded with overlay size of 10,000 and the following 10,000 random join and leave transactions are issued just like in the previous experiment. However some of these transactions are issued in parallel rather than serially. The client simulator ensures there are always a fixed number of concurrent join or leave transactions at any time. The batching timer is set to $10^6$ ms.
In Table 4.2, we can compare the average transaction time and throughput at different batching sizes. We find the average transaction time at 10,000 population increases comparing with the same population in Table 4.1. It is because the server handles many transactions simultaneously. With a batching size of 10, the transaction time is about 13% the non-batching mode time, and the server’s throughput increases about 180%. At some point, increasing the batching size actually increases transaction time and decreases throughput because the server has to wait until the number of queued transactions goes beyond the threshold or the timer expires. If we increase the number of concurrent transactions, it is easier to fill the batching queue, and the server can fire a batching operation. This is why in Table 4.2 300 concurrent transactions are better than 100 concurrent transactions at different batching sizes.

From the observations above, we know the batching mode may not be a good choice for an overlay with very few joins and leaves. In a very busy overlay network with frequent joins and leaves, an ATG AS in batching mode definitely performs more scalably.
CHAPTER FIVE

Summary

Without access control, overlay networks are vulnerable to suffering and even unwittingly participating in attacks. Existing solutions providing overlay security are either not scalable or focus on data encryption. In our ATG approach, the AS is only responsible for issuing credentials in response to join and leave requests. Overlay connectivity and data forwarding authorization are done locally with the AS intervention, fostering greater scalability. Because our ATG implementation uses a generic authorization interface, it is easily integrated with existing overlay, as we have demonstrated with PCE (Gitlin 2004) and YOID (Francis, Pryadkin, Radoslavov, and Govindan 2002).

According to the analysis of our experiments, the computation of join and leave at the server side is dominated by the signature operation. Both the computation and communication costs scale well with respect to overlay size. Furthermore, the performance of the AS can be improved significantly in the batching mode under heavy loads.

We find there are ideas that may enhance our work, and we list some of these below:

**Hash tree shrinking** In the current version of ATG protocol, the hash tree does not shrink. When a member leaves, its node in the hash tree is marked as unused until some new member is assigned to this node. The hash tree remains large, even if the overlay size decreases; consequently, there may be many wasted nodes in the hash tree. Shrinking a hash tree needs more complex protocol to carry details of tree structure, which may cost much more than the current method at the tree building stage.
**Message compression** In batching mode, when the overlay size is large, the broadcast message size limits the number of batched job. If the batching message is compressed, we can cache more transactions in one batching. Message compression may introduce some overhead both at the sender and receiver; therefore, compression algorithm complexity must be carefully considered.

**Recovery from stale** Currently, a stale member will try to get the missing message from its neighbor. If the neighbor does not cache this message, the stale member will send repair request to the server, which will increase the burden of the server. A better stale state recovery algorithm should minimize the repair requests to the server as well as the cache size at all members.
APPENDIX A

Signature operation comparison

RSA is very expensive in the cost of time. Encryption is more than three orders of magnitude slower than that of symmetric key cryptography. In this section, we compare several implementations of the RSA signature computation to see which one we should use in ATG. We know the C++ implementation is much fast than Java implementation. If the difference is enormous, we consider using C++ cryptography library through the Java Native Interface. This report includes the comparison results for the digital signature operation using Java, JNI, and C++. All tests use MD5 as the hash algorithm and RSA as the encryption algorithm. The key size is 1024 bits.

All tests were run on the machine with 2 Pentium 3 993MHz CPU’s and 1 GB RAM running on Linux kernel 2.4.20. The Java implementation used JCA (Java Cryptography Architecture). JNI used The Java Native Interface to call functions in CryptLib 3.0, which also provides signature service in C++ implementation. In each test, we input a random byte array and measured the total time for signature operation repeated a number of times. Then we changed the size of input array and repeat times to compare the difference of three implementations.

In Table A.1, we can see signature execution time of three implementations at different input size and repeating times. Generally speaking, the performance for JNI and C++ is better than Java implementation, and C++ is slightly better than JNI.

<table>
<thead>
<tr>
<th>Input Size (byte)</th>
<th>Repeat Times</th>
<th>Java (ms)</th>
<th>JNI (ms)</th>
<th>C++ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>1024</td>
<td>42.62</td>
<td>17.97</td>
<td>18.50</td>
</tr>
<tr>
<td>65535</td>
<td>1024</td>
<td>43.71</td>
<td>19.25</td>
<td>18.36</td>
</tr>
<tr>
<td>65535</td>
<td>2048</td>
<td>86.37</td>
<td>37.27</td>
<td>36.60</td>
</tr>
</tbody>
</table>
In the first test and the second, the change of data size to be signed does not introduce significant computation time. This is because that the encryption operation is only executed on the hash value from the output of MD5, which is a very fast operation. The speed of JNI and C++ is about twice faster than Java. We realize that this possible performance improvement is not significant enough for us to introduce JNI and a third party library to our system due to its large size and complexity to run and configure. Furthermore, Java code is platform independent. If C++ implementation is introduced, we must provide different versions for each OS. Therefore, we decide to use Java signature implementation in our ATG implementing to keep the integrity of the whole system.
BIBLIOGRAPHY


