Experimenting with Admission Control in P2P

Nitesh Saxena, Gene Tsudik, Jeong Hyun Yi
Computer Science Department
University of California at Irvine
USA
{nitesh,gts,jhyi}@ics.uci.edu

Abstract— **Peer-to-peer (P2P) security has received a lot of attention as of late. Most prior work focused almost entirely on issues related to secure communication, such as key management and peer authentication. However, an important pre-requisite for secure communication – secure peer admission – has been neither recognized nor adequately addressed. Only very recently, some initial work began to make inroads into this difficult problem. In particular, [1] constructed a peer group admission control framework based on various admission policies matched with appropriate cryptographic techniques. Recent results [2], [3] also illustrate the design of, and experiments with, certain group admission control mechanisms.**

In this work, we report on the implementation of Bouncer, an experimental peer group admission control toolkit used in [2] and its trial integration with two peer group systems with very different goals and semantics: Gnutella and Secure Spread. We also discuss some outstanding issues, challenges and future research directions relevant to this topic.

I. Introduction

The rising popularity of P2P applications prompts the need for specialized P2P security services and mechanisms. This has been recognized by the research community, however, the bulk of prior work is concerned with secure P2P communication, e.g., authentication, anonymity and key management. Although these issues are certainly important, another equally important topic has remained mostly unaddressed. Informally, it has to do with how one becomes a peer in a P2P system. More concretely, the technology for secure admission of peers into a P2P application simply does not exist. This statement does not contradict the fact that there are many currently operating P2P applications; they either operate in a completely open manner (i.e., have no admission control whatsoever) or admit peers on some *ad hoc* basis. This state of affairs bears a certain similarity to the early days of group key management when group keying was either non-existent or obtained by out-of-band means. To exploit this a little further, we observe that, just as trivial key management solutions severely limited the functionality of peer group applications, equally trivial admission control techniques will do (or already have done) the same. In other words, we believe that – without a well-thought-out architecture and appropriate techniques for peer admission – most P2P systems will sooner or later hit the proverbial "brick wall".

A. Prior Work

Recently, Kim, et al. [1] developed a group admission control framework based on various cryptographic techniques. This framework classifies group admission policy according to the entity (or entities) that makes peer admission decisions. The classification includes simple admission control policies, such as static ACL(Access Control List)- or attribute-based admission, as well as admission based on the decision of some fixed entity: external (e.g., a TTP) or internal (e.g., a group founder). Such simple policies are relatively easy to support and do not present much of a technical challenge. However, they are inflexible and ultimately unsuitable for a dynamic P2P setting. Static ACLs enumerate all possible members and hence cannot support truly dynamic membership (although they work well for closed groups). Admission based on decisions of a TTP or a group founder violates the peer nature of P2P, since the entire philosophy of P2P paradigm is based on collective, distributed services and decisions.

To address more challenging collective (group-centric) admission policies, a follow-on work [2] built upon the framework in [1] by designing a menu of suitable distributed mechanisms on a number of cryptographic techniques. This work yielded mechanisms for both centralized and (more challenging, yet also more realistic) decentralized group settings. In the latter, all current group members can take part in the admission process in a fully distributed manner. This work also assessed the practicality of distributed cryptographic mechanisms (such as verifiable threshold signatures) in both synchronous and asynchronous P2P settings. For an in-depth discussion of these admission control mechanisms, protocols and the experimental results, the reader is referred to [2], [3], [1].

In this work we focus on the design and implementation

of Bouncer, the admission control toolkit [2] integrated with an asynchronous P2P system (Gnutella [4]) and a synchronous group communication system with strong membership semantics (Secure Spread [5]). The Bouncer toolkit is general, i.e., it can be easily grafted onto any peer group setting.

II. Background

In this section, we describe a typical P2P admission procedure. The goal of this procedure is to allow a prospective member to obtain a group membership certificate. Using this certificate, a new member can prove membership and take part in future admission decisions.

As described in [2], the admission process is similar to a general voting mechanism whereby a prospective member needs to collect a certain minimum (threshold) number of positive votes (endorsements) before becoming a group member. There are two types of threshold admission policies: fixed and dynamic. The former is specified as the minimum number of votes, whereas, a dynamic threshold is specified as a fraction or percentage of the current group size. A fixed threshold is essentially a t-out-of-n model where the threshold t is fixed and n (current group size) varies over time. In contrast, a dynamic threshold (such as 30%) implies that t shrinks or grows in tandem with n.

The table below summarizes the notation used in the remainder of the paper.

TABLE I
NOTATION SUMMARY

TD	trusted dealer
GAuth	group authority
n	total number of peers
t	threshold ($t \leq n$)
M_{new}	prospective member
M_i	current member ($0 < i \leq n$)
PKC_{new}	public key certificate of M_{new}
GMC_{new}	group membership certificate of M_{new}

The "generic" peer admission process is as follows:

Step 0. Bootstrapping: A prospective peer M_{new} obtains the *group charter* [1] out of band and then the information of current group size from either GAuth or some bootstrap node. The group charter contains various parameters and admission policies, including: group name, signature/encryption algorithm identifiers, threshold (numeric or fractional corresponding to fixed or dynamic threshold, respectively), below-threshold policy and other optional fields. This process is performed only once per admission.

Step 1. Join Request: As shown in Fig. 1, M_{new} initiates the protocol by sending a join request (JOIN_REQ) message to the group. This message, signed by M_{new}, includes M_{new}'s public key certificate (PKC_{new}) and the target group name. How this request is sent to the group is application-dependent.[1]

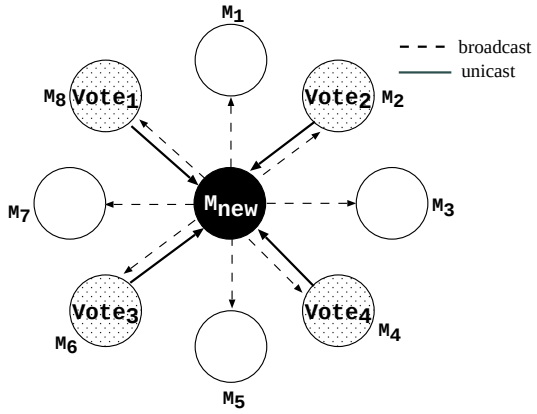


Fig. 1. Admission Control

Step 2. Admission Decision: Upon receipt of JOIN_REQ, a group member first extracts the sender's PKC_{new} and verifies the signature. If a voting peer approves of admission it replies with a signed message (JOIN_COMMIT) Several signature schemes (as described later in this section) can be used for this purpose. M_{new} verifies each vote.

Step 3. GMC Issuance: Exactly who issues the GMC_{new} for M_{new} depends on the security policy. If the policy stipulates using an existing GAuth, once enough votes are collected (according to the group charter), M_{new} sends to the GAuth a group certificate request message (GMC_REQ). It contains: PKC_{new}, group name, and the set of collected votes. In a distributed setting with no GAuth, M_{new} verifies the individual votes, and, from them, composes her own GMC_{new}.

Armed with a GMC, M_{new} can act as a *bona fide* group member. To prove membership to another party (within or outside the group) M_{new} simply signs a message (challenge) to that effect.

To carry out the admission decision process, various signature schemes are used, namely the plain RSA, Threshold RSA (TS-RSA) [6], [7], [8], Threshold DSA (TS-DSA) [9] and Accountable Subgroup Multisignatures (ASM) [10]. For a detailed description of these signature schemes and the admission protocol, refer to [2] and [3].

[1] Note that PKC_{new} does not have to be an identity certificate; it could also be a group membership certificate for another group.

III. BOUNCER: ADMISSION CONTROL TOOLKIT

We have implemented Bouncer, a general-purpose toolkit for P2P admission control based on the description in Section II. All cryptographic functions are developed using the OpenSSL library [11]. The toolkit is written in C on Linux and currently consists of about $45,000$ lines of code. The source code for the membership control toolkit is publicly available at [12].

A. System Design

The admission control system is made up of three basic layers of the architecture; GAC APIs, security and management services, and the underlying cryptographic functions. Figure 2 illustrates the architecture.

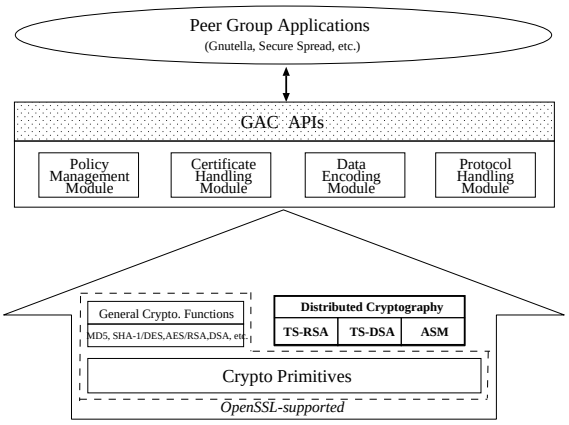


Fig. 2. GAC System Architecture

The GAC APIs define the application programming interface for accessing the admission control services. These APIs are useful when integrating our Bouncer with other peer group applications. The security and management services are carried out by the following modules:

- Policy Management Module
- Certificate Handling Module
- Data Encoding Module
- Protocol Handling Module

All security services are provided by the underlying cryptographic libraries.

B. Cryptographic Libraries

Most of the general cryptographic functions such as SHA-1, RSA, DSA, and so on, are supported by OpenSSL. Specifically, we have implemented three distributed cryptographic schemes on top of OpenSSL, and embedded our libraries into it. The Bouncer supports four different signature schemes; plain RSA, ASM, TS-RSA, and TS-DSA as addressed earlier.

C. Security and Management Services

1 Policy Management Module

A *policy management* module is the component which checks for conformance to the policy specified in the *group charter* [1]. First, this module contains functions to check the threshold type. If the threshold type is a static, it checks if the number of current members is at least equal to the threshold ($n \geq t$). If $n < t$, the policy manager enforces the `BelowThreshold` policy which requires it to either forward the `JOIN_REQ` to GAuth directly, or to reset the threshold to reflect current n.

In most P2P systems, group size can fluctuate drastically within a short time. As the number of peers grows or shrinks, we need to increase or decrease the threshold. Since updating the threshold is an expensive operation which requires a random number generation, it is impractical for every membership event to trigger an update process. In order to prevent this, we apply a simple *window* mechanism as shown in Fig. 3. Specifically, every member keeps state of n_{old}, which is the group size at the time of the last threshold-update process. A new threshold-update process is triggered only when the difference between the current group size n_{cur} and n_{old} is greater than Win – the window buffer. In other words, threshold update process is triggered only when $|n_{cur} - n_{old}| \geq Win$.

Function **GAC_Dynamic_Threshold_Update();**

Input parameters:

```
X509* GChart,          \* group charter *\
int N_old,             \* old group size *\
int N_cur,             \* current group size *\
int T_cur              \* current threshold *\
```

Body:

```
int diff;
int offset;
int Win;               \* Window buffer size *\
int T_new              \* new threshold *\

T_new=T_cur;
Win=WIN_TIMES*GChart.threshold.fixed;
diff = N_cur - N_old;
if (diff >= Win) {
   offset = ⌊diff / Win⌋;
   N_old = N_old + (offset*Win);
   T_new = ⌊(GChart.threshold.dynamic /
            100) * N_old⌋;
   if (T_new > T_cur)
       T_cur=T_new;
}
return T_new;
```

Fig. 3. Dynamic Threshold Update Procedure

2 Certificate Handling Module

Both GMC-s and group charters generated by the Bouncer are compatible with X.509v3 [13]. This *certifi-*

cate handling module takes care of all functions related to certificate compatibility. For example, in the group charter, we need to define several attributes in the extension field of certificate in order to codify certain admission policy. And this module also has a function to bind the identity of GMC to that of PKC as shown in Fig. 4 to protect against the Sybil attack [14], assuming that a Certification Authority (CA) issues a PKC with a unique identity to each user.

Further, possession of a GMC does not prove that the GMC actually belongs to the bearer. One way to accomplish this is by requiring for every group member to have a standard X.509 public key certificate (PKC) issued by the CA. The GMC simply needs to contain the public key of the member extracted from her PKC. Now the member (bearer of a GMC) can prove ownership of the GMC by demonstrating knowledge (e.g., by signing a message) of the private key corresponding to the public key referred to in in the GMC.

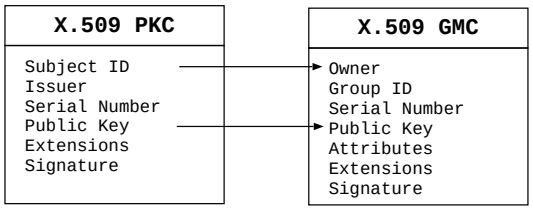


Fig. 4. Binding GMC to PKC

3 Data Encoding Module

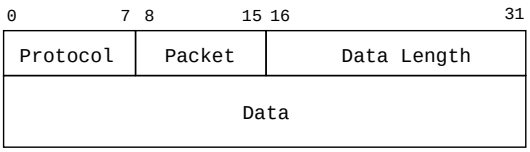
The *data encoding* module contains all encoding and decoding functions which convert ASN.1-formed messages to and from DER-encoded from. For example, `i2d_PS_Join_Request()` is a function which converts ASN.1-structured `JOIN_REQ` message based on plain RSA into DER-encoded binary data in order to transfer the message over the networks. Similarly, `d2i_PS_Join_Request()` is called when receiving `JOIN_REQ` message, to get internal form of the message.

4 Protocol Handling Module

The *protocol handling* modules includes functions used to identify admission control protocols and transfer the messages to and from the corresponding libraries. Fig. 5 shows the structure of GAC packet. Each packet is classified on the protocol using the packet type in the packet header.

D. GAC APIs

Application developers require no special knowledge of the organization of the security and management modules as well as cryptographic libraries. They just need




```
Protocol: the protocol identifier
   1) PS (0x01)
   2) TS-RSA (0x02)
   3) TS-DSA (0x03)
   4) ASM (0x04)
Packet: the packet type defined
   1) JOIN_REQ: the join request (0x01)
   2) JOIN_CMT: the join commit (0x02)
   3) CHAL_REQ: the challenge request (0x03)
   4) CHAL_RLY: the challenge response(0x04)
   5) SIGN_REQ: the sign request (0x05)
   6) PART_SIG: the partial signature reply (0x06)
   7) GMC_REQ: the GMC request (0x07)
   8) GMC_RLY: the GMC reply (0x08)
```

Fig. 5. GAC Packet Structure

to use the GAC function interface to bulid any application. GAC APIs are logically partitioned into functional categories. The goal of this logical partitioning is to assist application developers in understanding and making effective use of the security APIs. With this logical classification, we support the following APIs. Among these APIs, `GMC_Request()` and `GMC_Reply()` are optionally required only when we can assume the presence of a centralized authority.

- Plain RSA APIs

```
GAC_PACKET *PS_Join_Reqest();
GAC_PACKET *PS_Join_Commit();
GAC_PACKET *PS_GMC_Request(); /* optional */
GAC_PACKET *PS_GMC_Reply();   /* optional */
```

- TS-RSA APIs

```
GAC_PACKET *TSS_Join_Request();
GAC_PACKET *TSS_Join_Commit();
GAC_PACKET *TSS_Sign_Request();
GAC_PACKET *TSS_Part_Sign();
GAC_PACKET *TSS_GMC_Request(); /* optional */
GAC_PACKET *TSS_GMC_Reply();   /* optional */
```

- TS-DSA APIs

```
GAC_PACKET *TSD_Join_Request();
GAC_PACKET *TSD_Join_Commit();
GAC_PACKET *TSD_Chal_Req();
GAC_PACKET *TSD_Chal_Rly();
GAC_PACKET *TSD_Rnd_Req();
GAC_PACKET *TSD_Rnd_Rly();
GAC_PACKET *TSD_Sign_Request();
GAC_PACKET *TSD_Part_Sign();
GAC_PACKET  *TSD_GMC_Request(); /* optional */
GAC_PACKET *TSD_GMC_Reply();    /* optional */
```

- ASM APIs

```
GAC_PACKET *ASM_Join_Request();
GAC_PACKET *ASM_Join_Commit();
GAC_PACKET *ASM_Sign_Request();
GAC_PACKET *ASM_Part_Sign();
GAC_PACKET *ASM_GMC_Request(); /* optional */
GAC_PACKET *ASM_GMC_Reply();   /* optional */
```

IV. Integration with P2P and Group Communication Systems

To evaluate the performance of our mechanisms and to measure the overhead incurred due to incorporating admis-

sion control in the context of real-world application, we integrated the Bouncer with a popular P2P file sharing system, Gnutella and with a wide area secure group communication system, Secure Spread. Secure Spread is selected as an example of a synchronous P2P system, and Gnutella as an asynchronous one. We integrated the centralized admission protocol with the former and the decentralized one with the latter to measure the performance in both settings.

In the following sub-sections, we discuss the implementation details for the integration with both the systems.

A. Integration with Gnutella

The *Gnutella* is the "pure" P2P file sharing system which is closest to the ideal structure of the P2P spirit, where all participants have uniform role. In such an architecture, users are free to join and leave the group. Even malicious users can easily join to deny or disrupt the system. To prevent such a security threat in a fully distributed P2P environment, we integrated our Bouncer with Gnut-0.4.21 [15] (an open-source Gnutella [4] implementation).

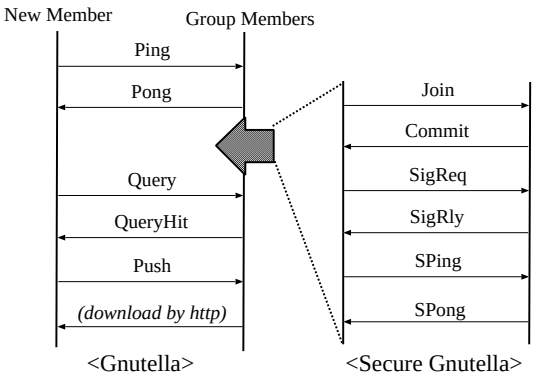


Fig. 6. Secure Gnutella Protocol Flow

At the setup phase of the Gnutella protocol, a connection is established by communicating so-called `Ping` and `Pong` messages which are based on IP addresses as shown in Fig. 6. To look for a file, a new member sends out a broadcast `Query` message to every member to which it is directly linked. The group members identifying the requested file in their repository answer with a `QueryHit` message which is returned to the connection from which the request arrived. The `QueryHit` message contains the *ResultSet* and the pair $\langle$*IP address, port*$\rangle$ that must be used to download the file via HTTP.

The *Secure Gnutella* protocol, illustrated in Fig. 6, defines some extra messages for secure admission control; `Join, Commit, SigReq, SigRly, SPing,` and `SPong`. The message format for new protocol steps is defined as follows;

- `Join(mesg, PKC, Sig)`
- `Commit(port, IP addr, GMC, commit_val, Sig)`
- `SigReq(servant ID, sigreq_val, Sig)`
- `SigRly(servant ID, sigrly_val, Sig)`
- `SPing(Group ID, GMC)`
- `SPong(port, IP adddr, # of files, # of Kbytes, GMC)`

First, like in a standard Gnutella protocol, a new member broadcasts to all her neighbors `Join` message which contains the join request message and her own PKC. Upon reception of the `Join` message, some of group members reply with `Commit` message to confirm that they will participate in admission process. In this message, the `commit_val` is an encapsulated message of the GAC protocol, which is DER-encoded form. The `SigReq` and `SigRly` are newly specified messages for the GAC protocol. For checking the integrity of protocol message, `Commit, SigReq`, and `SigRly` messages include the signature thereon which is PKCS7-formatted.

In order to prevent Sybil attacks [14], we modified standard `Ping` and `Pong` messages so that the connection is made only if the responder answers with its valid GMC. For this purpose, we specified two new messages: `SPing` and `SPong`. The `SPing` message contains the requester's PKC, and the `SPong` message contains the responder's GMC and its signature (to prove possession of its private key). In *Secure Gnutella* system, standard `Ping` and `Pong` messages are no longer used.

B. Integration with Secure Spread

Spread [16] is a wide area group communication system. It provides a high performance messaging service that is resilient to faults across external or internal networks. Spread functions as a unified message bus for distributed applications, and provides highly tuned application-level multicast and group communication support. Spread services range from reliable message passing to fully ordered messages with delivery guarantees, even in case of computer failures and network partitions.

Secure Spread [5] is an application built atop Spread. It enhances Spread by integrating security services and key management.

In its present form, Secure Spread supports only static group access control which is provided at the daemon level using ACL's. This clearly poses a single point of failure problem. Moreover, as argued before, static admission control is no good for dynamic groups. Secure Spread also has a notion of a *flush* mechanism, in which all current group members need to acknowledge any change in membership (e.g. join, leave, partition, merge). A prospective member can join a group only after it has received *flush OK* messages from all current group members. This is a very weak form of providing admission as this mechanism of-

fers no security at all because there involves no authentication of either prospective or current members. Moreover, all group members need to be involved in every admission process simultaneously.

In order to resolve these problems and of course to measure the performance, we integrated Bouncer with Secure Spread. The integration involves extension to the Spread API and can be used with any application (including Secure Spread) that uses Spread.

We added the following function to the current interface of Spread.

```
int SP_GAC_join(mailbox mbox, const char *group)
```

This function is declared in `sp.h` of Spread source tree. It joins a group using the group admission mechanisms described in previous sections, with the name passed as the string group. If the group does not exist among the Spread daemons it is created, otherwise it joins the existing group. The mbox of the connection upon which to join a group is the first parameter. The group string represents the name of the group to join.

The function Returns 0 on success or one of the following errors (< 0):

`ILLEGAL_GROUP`
The group given to join was illegal for some reason. Usually because it was of length 0 or length $>$ `MAX_GROUP_NAME`.

`ILLEGAL_SESSION`
The session specified by mbox is illegal. Usually because it is not active.

`CONNECTION_CLOSED`
During communication errors occured and the join could not be initiated.

In case, the prospective member is not able to receive enough votes, the function call will not be completed and the member will wait forever.

`JOIN_REQ` message is encapsulated within the standard spread message and sent to all the group members using Spread multicasting. Fig. 7 and 8 show Spread header and the encapsulation of GAC message inside the spread message (sizes are in bytes). The function makes a call to the `SP_multicast` function of the Spread API. For details regarding the multicast message, refer to the spread function interface in [16].

In order to receive replies back from the group members, the function `SP_GAC_join()` uses the `SP_receive` function of the Spread API.

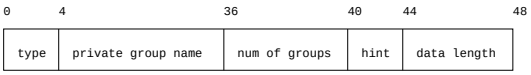


Fig. 7. Spread Message Header

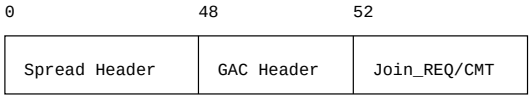


Fig. 8. Spread GAC Encapsulation

We have also modified the `SP_receive` function. This takes care of the fact that when a current group member receives the `JOIN_REQ` message from a prospective member, it responds with a `JOIN_CMT` message as its vote. This message again is encapsulated within the standard Spread message and its sent to the requesting member using the Spread unicasting. For this purpose, we again use the `SP_multicast` function to send unicast message to the new member using its private group name which is represented by *#private user name#daemon name*.

After collecting enough votes from group members, the prospective member requests the GMC from the external GAuth. Once, the GAuth issues the GMC to the new member, the admission process is completed. Then, the spread daemons update the membership information and update/distribute the new key to the newly joined member.

V. Experiments

In our experiments with Gnutella and Secure Spread, we measured the costs of basic operations and then compared the performance of four cryptographic protocols with both fixed and dynamic thresholds. We used 1024-bit modulus in all mechanisms; that is, 1024-bit N in RSA and TS-RSA, and 1024-bit p and 160-bit q in TS-DSA and ASM.

Since each protocol has different number of communication rounds, we measured total processing time from sending of the `JOIN_REQ` to obtaining new GMCs[2]. This means the join cost includes not only the signature generation and verification time in basic operations, but also the communication costs such as packet encoding/decoding time, the network delay, and so on. To get reasonably correct results, the experiments were repeated more than 1000 times for each.

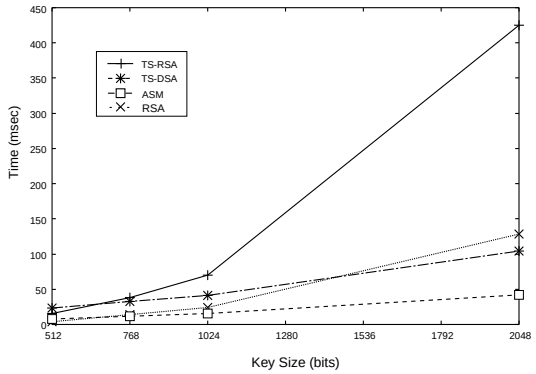
A. Computation Costs

In this section, we demonstrate the cost of each signature scheme used as a primitive in Bouncer.

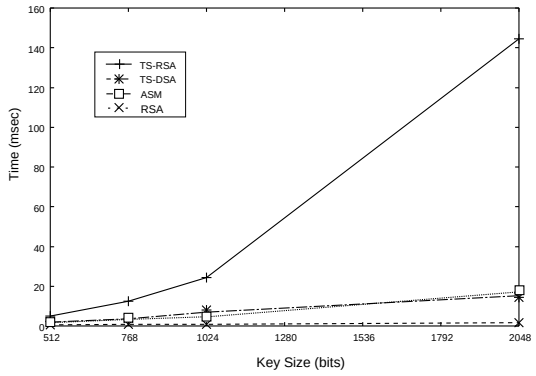
Fig. 9(a) shows the cost of signature generation versus the key size, where t=3. We found that in TS-RSA, the cost

[2]In these experiments we did not consider the *partial share shuffling* for both TS-RSA and TS-DSA.

in generating a signature is much more expensive than that of RSA signature generation, since we can not apply *CRT (Chiness Remainder Theorem)* to speed up the compuation as in plain RSA scheme. TS-RSA is slightly better than TS-DSA with 512-bit modulus, while TS-DSA is faster than TS-RSA with larger key size. As evident from the figure, ASM is the best performer because it is based on the efficient Schnorr's signature scheme.



(a) Signature Generation



(b) Siganture Verifi cation

Fig. 9. Basic Operation Cost

Fig. 9(b) shows the cost of signature verification with varying key sizes. In PS, the cost of signature verification is proportional to the threshold. All other schemes, except PS, have only one resulting signature due to the aggregation of partial signatures. We also observe that the verification costs of TS-DSA and ASM are almost the same as for the underlying DSA and Schnorr signature schemes respectively. However, verification cost for TS-RSA is extremely high. This is because $m^N \pmod N$ in *t-bounded offsetting algorithm* [6] has to be computed almost every time the signature is verified. Due to this expensive operation, it turns out that the TS-RSA performs much worse than the other schemes, contrary to our intuition.

B. Signature Size

From the analysis of the computation cost above, it turned out that both plain RSA and ASM are more efficient than the two threshold signature schemes. However, the length of the signature in plain RSA and ASM is linear in threshold t. In this experiment we extract the identities (which are X.509 DN formatted) from the GMC-s. We also used 1024-bit RSA key and SHA-1 as a hash function for both ASM and TS-DSA.

In both plain RSA and ASM schemes, the signers' identities should be included in the resulting signature. Due to the size of the identity (i.e., 952 bits), the resulting signatures become very large depending on the threshold; whereas, both TS-RSA and TS-DSA have a constant signature size (i.e., 1024 bits and 320 bits, respectively). For example, from the Fig. 10, we can see that the size of plain RSA is about 150 times as long as that of ASM when the threshold is set to 25. Therefore, we recognize that both plain RSA and ASM would not be suitable for large groups where the bandwidth is a major concern.

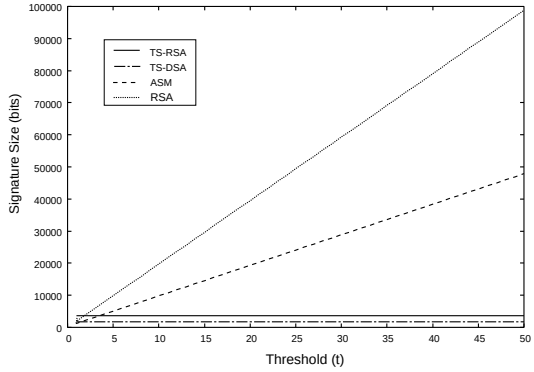


Fig. 10. Signature Size

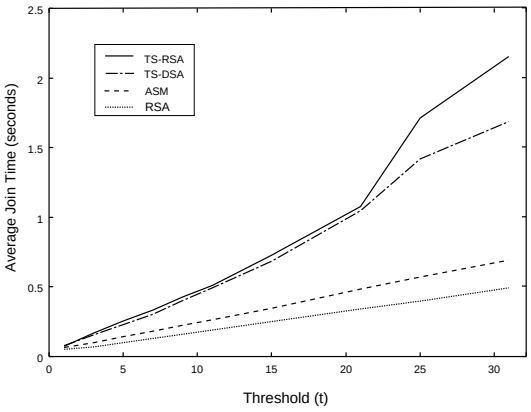
C. Gnutella Experiments

We measured the performance of the `Secure Gnut` which is the *Gnut* system integrated with our Bouncer. We performed all measurements on the following Linux machines connected with a high-speed LAN: P4-1.2GHz, P3-977MHz, P3-933MHz, and P3-797MHz.

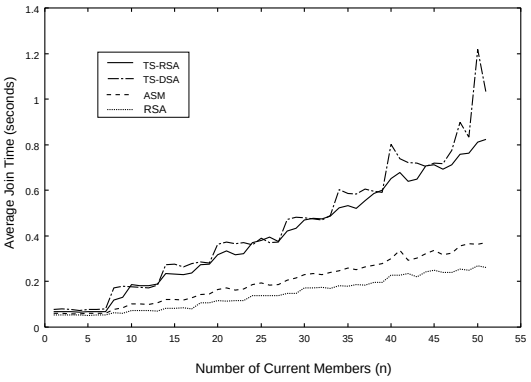
Fig. 11(a) shows the join cost for the static threshold case. Fig. 11(b) shows the join costs for the dynamic threshold case where the threshold ratio is set to 30% of current group size. All of these measurements were performed with the equal number of member processes on each machine.

D. Secure Spread Experiments

For our experiments with Secure Spread, we used a cluster of 10 machines at Johns Hopkins University. Each ma-



(a) Fixed Threshold



(b) Dynamic Threshold (R=30%)

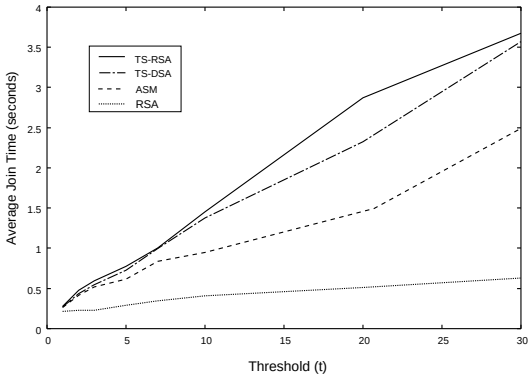
Fig. 11. Gnutella Expriments

chine has P3-667 MHz CPU, 256 KB Cache and 256 MB memory and runs Linux 2.4. We ran Spread daemons on all machines which formed a Spread Machine Group. Almost equal number of clients running on these machines connect randomly to the daemons. The new joining member is a client running on a machine at UC Irvine with a Celeron 1.7 GHz CPU, 20 KB cache and 256 MB memory.

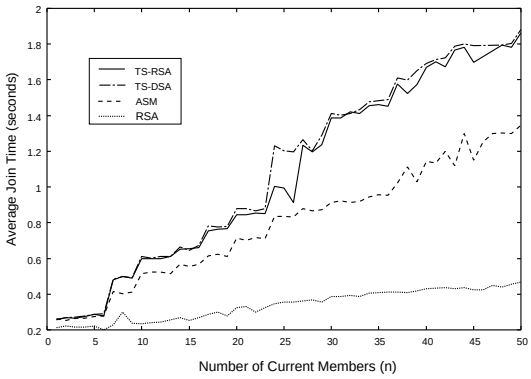
Experiments were performed with the above testbed for both fixed and dynamic thresholds for all signature schemes discussed thus far.

Fig. 12(a) shows the plot for the average time taken by a new member to join a group with a fixed threshold. We performed this test with 4-5 processes on each machine and measured the join cost by changing the threshold. As expected, plain RSA is the best performer in terms of computation time. However, we also see that both TS-RSA and TS-DSA exhibit reasonable costs ($<$ 1 sec.), at least until t=10.

Fig. 12(b) show the plots for the average time for a new member to join a group with a dynamic threshold. In this experiment, the threshold ratio (R) is set to 30% of the current group size. The actual numeric threshold is determined by multiplying the group size by R. We measured the performance up to $n = 50$.



(a) Fixed Threshold



(b) Dynamic Threshold (R=30%)

Fig. 12. Secure Spread Expriments

For a detailed discussion regarding the results of these experiments, the reader is referred to [2].

VI. Discussion

As it is clearly reflected from the measurement results above, all the advanced cryptographic constructs i.e. the threshold signatures and the multisignatures perform quite poorly. Especially the threshold signatures are about 4-7 times costlier than the plain RSA signatures for relatively larger groups. But, as discussed in [2], since for plain signatures and multisignatures the size of the combined signature and thus the size of the GMC varies proportionally with the threshold, we can't pick just one signature scheme for all P2P settings. A certain balance has to be maintained between the size of the GMC and the average join cost apart from the choice of the scheme-specific features like anonymity, accountability, membership awareness and so on.

One might argue that group signature scheme [17] might also be a possible candidate for the admission control especially in a P2P scenario where signer anonimity is a must. We did in fact implement the group signature scheme in our toolkit and experimented with it. But, unfortunately, we have to rule out the possibility of using group signatures as they perform way worse than the other signature schemes. Moreover, group signature scheme can only be used for the centralized admission protocol as it requires the presence of a group manager.

In summary, we are faced with a couple of challenges in order to provide secure admission control. One challenge is to make the admission process as distributed as possible and the other is to do so in a highly efficient manner with the lowest possible overhead (storage as well as bandwidth). Though in a P2P setting, a distributed approach seems like the most natural one but it turns out to be the hardest as well. A admission control mechanism will only be applicable in mobile ad-hoc and sensor networks if it is both distributed and power-efficient. As of now none of the schemes seem very useful in these scenarios.

VII. Future Directions

As is evident from the experimental results and above discussion, there is a lot of scope for improvement and promise for further work. We have seen that there is a tradeoff between the performance and the signature size among various schemes. So, one immediate objective is to find/design an efficient signature scheme which on one hand has a fewer rounds in the protocol and on the other smaller signature size in the GMC. Recently proposed *a*ggregated signature scheme [18] appears to be an attractive candidate for the same. But, we claim that one particular signature scheme would not be sufficient for our purpose of admission control. The choice of the scheme to be used has to be made based on a number of factors like type/size of group, bandwidth, various features desired and the group policies.

Another possible enhancement could be to have admission decision based on a trust based model. In the usual more practical scenario, a group member can only probabilistically vote in or vote out a prospective member. In the presence of a trust model, voting would be more deterministic.

This work uses a certificate based approach towards admission control. With certificates arises the issue of revocation which could be a hard problem to deal with in a distributed setting. In order to avoid this issue, another future direction is to design a non-certificate based approach for admission.

Another prospect of future work is the complementary problem of membership revocation. If providing secure admission is hard, solving the problem of revocation will be even more challenging.

References

[1] Yongdae Kim, Daniele Mazzocchi, and Gene Tsudik, "Admission Control in Peer Groups," in *IEEE International Symposium on Network Computing and Applications (NCA)*, Apr. 2003.

[2] Nitesh Saxena, Gene Tsudik, and Jeong Hyun Yi, "Admission Control in Peer-to-Peer: Design and Performance Evaluation," in *ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN)*, October 2003, pp. 104–114.

[3] Maithili Narasimha, Gene Tsudik, and Jeong Hyun Yi, "On the Utility of Distributed Cryptography in P2P and MANETs: The Case of Membership Control," in *IEEE International Conference on Network Protocol (ICNP)*, November 2003, pp. 336–345.

[4] The Gnutella Protocol Specification v0.4, ," `http://www.clip2.com/GnutellaProtocol04.pdf`.

[5] Secure Spread Project, ," `http://www.cnds.jhu.edu/research/group/secure_spread/`.

[6] Jiejun Kong, Petros Zerfos, Haiyun Luo, Songwu Lu, and Lixia Zhang, "Providing Robust and Ubiquitous Security Support for MANET," in *IEEE 9th International Conference on Network Protocols (ICNP)*, 2001.

[7] Haiyun Luo, Petros Zerfos, Jiejun Kong, Songwu Lu, and Lixia Zhang, "Self-securing Ad Hoc Wireless Networks," in *Seventh IEEE Symposium on Computers and Communications (ISCC '02)*, 2002.

[8] Jiejun Kong, Haiyun Luo, Kaixin Xu, Daniel Lihui Gu, Mario Gerla, and Songwu Lu, "Adaptive Security for Multi-level Ad-hoc Networks," in *Journal of Wireless Communications and Mobile Computing (WCMC)*, 2002, vol. 2, pp. 533–547.

[9] R. Gennaro, S.Jarecki, H.Krawczyk, and T.Rabin, "Robust Threshold DSS Signatures," in *EUROCRYPT '96*, Ueli Maurer, Ed. IACR, 1996, number 1070 in LNCS, pp. 354–371.

[10] Kazuo Ohta, Silvio Micali, and Leonid Reyzin, "Accountable Subgroup Multisignatures," in *ACM Conference on Computer and Communications Security*, November 2001, pp. 245–254.

[11] OpenSSL Project, ," `http://www.openssl.org/`.

[12] Peer Group Admission Control Project, ," `http://sconce.ics.uci.edu/gac`.

[13] R. Housley, W. Polk, W. Ford, and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 3280, IETF, Apr. 2002.

[14] John R. Douceur, "The Sybil Attack," in *International Workshop on Peer-to-Peer Systems (IPTPS'02)*, March 2002.

[15] Gnut v0.4.21 source code, ," `http://schnarff.com/gnutelladev/source/gnut`.

[16] Spread Project, ," `http://www.spread.org/`.

[17] Giuseppe Ateniese, Jan Gamenisch, Marc Joye, and Gene Tsudik, "A Practical and Provably Secure Coalition-Resistant Group Signature Scheme," in *CRYPTO '00*, Mihir Bellare, Ed. IACR, 2000, number 1880 in LNCS, pp. 255–270.

[18] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps," in *EUROCRYPT '03*, Eli Biham, Ed. IACR, 2003, number 2656 in LNCS, pp. 416–432.