The Wisdom of Crowds as a Model for Trust and Security in Peer Groups

by

Justin D. Whitney

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

September 2005

APPROVED:

_____

Professor Fernando C. Colon Osório, Thesis Advisor

_____

Professor Murali Mani, Reader

_____

Professor Michael Gennert, Head of Department

Abstract

Traditional security models are out of place in peer networks, where no hierarchy exists, and where no outside channel can be relied upon. In this nontraditional environment we must provide traditional security properties and assure fairness in order to enable the secure, collaborative success of the network. One solution is to form a Trusted Domain, and exclude perceived dishonest and unfair members.

Previous solutions have been intolerant of masquerading, and have suffered from a lack of precise control over the allocation and exercise of privileges within the Trusted Domain. Our contribution is the introduction of a model that allows for controlled access to the group, granular control over privileges, and guards against masquerading. Continued good behavior is rewarded by an escalation of privileges, while requiring an increased commitment of resources. Bad behavior results in expulsion from the Trusted Domain. In colluding with malicious nodes, well behaved nodes risk losing privileges gained over time; collusion is thereby discouraged.

We implement our solution on top of the Bouncer Toolkit, produced by Narasimha et al. [25], as a prototype Peer to Peer file sharing network. We make use of social models for trust [24] [30] [26], and rely on new cryptographic primitives from the field of Threshold Cryptography. We present the results of an experimental analysis of its performance for a number of thresholds, and present observations on a number of important performance and security improvements that can be made to the underlying toolkit.

Acknowledgments

Thank you to my advisor Professor Fernando C. Colon Osório for the tremendous opportunity, and the fantastic ride. Thank you to my Reader Professor Murali Mani for his hard work. Finally, thank you to everyone at the WPI Secure Systems Research Laboratory (WSSRL) for their ideas.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Peer networks exhibit several interesting properties that make the application of traditional security techniques difficult [20]. Despite this difficulty, the application of some technique is necessary if the peer group is to operate securely. We begin by examining the environment of a peer network.

In a peer group, a number of network hosts collaborate to achieve some collective goal. In a Mobile Ad-Hoc Wireless Network (MANET), for example, where no access point is present and therefore wireless nodes act as both senders and routers of packets, each wireless host collaborates to provide the basic network services of routing and traffic forwarding for the group [31]. Wireless hosts achieve this by forming a Peer-to-Peer (P2P) network. Another example, with greater significance to this thesis, is a P2P File Sharing network such as Gnutella [8]. In this case, lacking any central point of control or authority, hosts collaborate to exchange files, again forming a P2P network to do so. In both examples, there is a need for fair, secure collaboration to achieve the common goal.

Within a peer group, no hierarchy is initially assumed, and all nodes are given equal authority. Furthermore, no channel to nodes outside of the peer network is assumed to exist. In particular, these assumptions make it difficult to use traditional Public Key (PKI)

[12] techniques in this setting. Given the lack of an outside channel, no Certification Authority (CA) can be reliably made use of to determine authenticity. Given the lack of a hierarchy, no single host can be given a fixed role in the network, such as that of an acting CA for the group; trust would have to be arbitrarily placed in such a host, as there is no means of initially assessing whether it is more or less trustworthy than any other node in the group. Furthermore, such an assignment creates a single point of failure.

Finally, a peer group should operate in an ad-hoc manner, integrating new nodes without prerequisite. This further increases the challenge of designing security mechanisms suitable to such a group, as no preexisting Security Association (SA) can be made use of. In the case of the MANET, this means that lower layer security mechanisms such as WEP [13] may not be used, as they require a prior SA. This requirement, in combination with the lack of fixed roles and a CA, often makes it difficult to verify hosts as being unique, and causes masquerading to be a problem.

In the face of these many complexities, nodes must still uphold traditional security properties such as data integrity and authenticity in order for the network to function securely. Furthermore, nodes must act fairly, otherwise selfish nodes can degrade the result of collaboration. It is a unique challenge to implement security services and provide fairness in such an environment.

In this paper we present a new model that allows for the ad-hoc formation of peer groups, while providing traditional security properties, and assuring the fair operation of the network. Trust in honest members is increased over time, and privileges can be extended to trustworthy members with granularity. Dishonest members are detected and expelled from the group. Our model improves on existing work which limits trust to a binary relationship (all or none), and which does not allow for privileges to be issued with granularity. Furthermore, our model incorporates elements which can be used to discourage or even prevent masquerading, which is a considerable problem in prior mod-

els. We make use of social models for trust [30] [24], and rely on new cryptographic primitives from the field of Threshold Cryptography. We introduce these new materials in immediately following sections, and then proceed to describe our model.

1.1 Secret Sharing and Threshold Cryptography

Our work makes considerable use of new cryptographic primitives, primarily Threshold Cryptography. While reliance on a CA is not possible in a peer group given the constraints discussed in § 1, the use of Threshold Cryptography allows this role to be distributed between some or all members in such a way as to allow a subset of the group to perform signing in place of the CA. The principle cryptographic primitive used by our work is a threshold variation of the RSA algorithm which we refer to as TS-RSA [21]. The TS-RSA algorithm uses Shamir's Secret Sharing [29] technique to share an RSA private key amongst group members. This technique is based on Lagrange Polynomial interpolation [29]. We present these concepts concisely below, and refer the reader to the sources for further details.

In Shamir's secret sharing scheme [29], a secret S is broken into n pieces, any t of which may later collaborate to recover the secret. Such a system can be called a (t, N) secret sharing scheme. Some threshold cryptographic systems make use of techniques known collectively as *proactive secret sharing* which allow t to increase for each increase in N. We focus our attention here on static systems in which this is note the case, and refer the reader to Herzberg et al. [18] for further details. Initially, a trusted dealer chooses a large prime q and selects a polynomial $f(z)$ over $\mathbb{Z}_q$ of degree $t-1$ such that

$$f(0) = S \tag{1.1}$$

Each share of the secret $ss_i \in (ss_1, ss_2, ..., ss_n)$ is computed as in Equation (1.2) and is securely dealt to the node i.

$$ss_i = f(i) \mod q \tag{1.2}$$

Using the Lagrange Polynomial Interpolation formula in Equation (1.3) any t nodes may then recover the secret.

$$f(z) = \sum_{i=1}^{t} ss_i \cdot l_i(z) \pmod q \tag{1.3}$$

Any fewer than t collaborating nodes may not recover S and may gain no information about it. Each *publicly available* lagrange coefficient l_i is calculated as in Equation (1.4) [25].

$$l_i(z) = \prod_{j=1, j\neq i}^{t} \frac{z-j}{i-j} \pmod q \tag{1.4}$$

Many Threshold Cryptographic primitives, such as the Threshold RSA scheme presented by Kong et al. in [21], use an existing public-key algorithm, in this case RSA, where the secret S is the private key, which is distributed amongst n nodes. While not all Threshold Cryptographic systems allow it, in their work, Kong et al. propose a system by which any t nodes may issue *partial signatures* using the shared private key. These signatures may be reconstituted by the recipient into a full signature, signed by the group private key, on some message M. In this way, t nodes may make use of the shared private key S while never disclosing the group secret to any one node. We briefly present this work and refer the reader to [21] for further details.

To issue a partial signature on a message M, a node i treats its partial share ss_i as the exponent in the RSA algorithm, computing

$$M^{ss_i} \mod q \tag{1.5}$$

The recipient of t such partial signatures is able to combine them to form a full signature on M signed by the group private key; one obstacle remains to this reassembly, which we now address.

Given $f(0) = S$ (Equation (1.1)), we may restate Equation (1.3) as

$$d = S = \sum_{i=1}^{t} ss_i \cdot l_i(0) \pmod q \tag{1.6}$$

For some value, j, we can also say that $\sum_{i=1}^{t} ss_i \cdot l_i(0) \pmod q = j \cdot q + d$ [21]. However, there can be no mathematical assurance that $M^{j \cdot q + d} \equiv M^d \equiv M^S \pmod q$. Kong et al. overcome this problem through the use of what they term the *K-bounded coalition offsetting algorithm*[1]. This algorithm makes use of the group public key $PK =< e, n >$, and functions as follows.

Figure 1.1: K-Bounded Coalition Offsetting Algorithm

$Z = M^{-n} \mod q$
$l = 0$
while $l \leq K$ **and** $M \not\equiv Y^e \pmod q$ **do**
    $Y = Y \cdot Z \mod q$
    $l = l + 1$
done

Result: $Y \equiv M^d \mod q$

This algorithm ensures that the result of combining the partial signatures on M is equivalent to a full signature on M using the group private key. The use of partial signatures and the algorithm above let us assign the role of a CA to some subset of a peer group with a number of useful provisions. We can tolerate some number of faulty or malicious

[1] In this case $K = t$, the threshold.

nodes less than t, while still allowing the group CA to function as intended, and without compromising S.

Furthermore, because t nodes are *required* to sign for some message M in order to fully sign as S, we create a proxy for consensus that demands that t nodes *agree* to sign some message M. If entrance into the peer group were based on this proxy, for example, then by obtaining a group signature on some message M a node can prove that it was accepted for entrance. This idea of a threshold t as a proxy for consensus is explored in two relevant papers, each of which we now present briefly.

1.2 Prior Work

In their research (see [31]), Yang and colleagues sought a solution that secured routing and packet forwarding in a MANET. In their solution nodes secure those two services through the formation of a Trusted Domain (TD) within the peer network. Within the TD, members are monitored promiscuously and a (t, N) threshold-shared private key is used to certify fair and honest nodes by signing *tokens* for each member. Each token is signed only if t neighboring nodes have observed that some node i has not acted maliciously, having routed and forwarded packets as required. Members acting unfairly or maliciously are expelled when their tokens expire and no coalition of t neighbors signs a new token. Data integrity and authenticity between nodes is upheld through the use of private keys held by each node.

Narasimha et. al. (see [25]), further explored the concept of trust in a MANET. Specifically, they explored the problem of controlling admission of nodes to a Trusted Domain such as the one introduced by Yang et al. Their work abstracted the problem of safeguarding membership in a Trusted Domain, creating a framework that could express the requirements for membership in a general way. Their implementation of this framework,

which they termed the *Bouncer Toolkit*, implemented a number of threshold cryptographic primitives. An arbitrary peer group can be implemented on top of the toolkit by selecting the appropriate signature scheme and membership requirements. We introduce a new model that makes use of this framework, benefiting from the ideas presented in both of these papers.

Prevention of masquerading proved to be a shortcoming of both the work of Narasimha et al. and Yang et al. In their work, Yang et al. [31] chose to use the MAC address of a wireless node was used to ensure that no one node received more than one share of the group secret, or was able to sign more than once on a token for some node. The MAC is known to be easily forgeable, however, and should not be used for this purpose. In Narasimha et al. a certificate signed by a CA was required of each node applying for membership to some peer group. While this offloaded the problem of masquerading on to the CA, it created a requirement for admission that invalidated the possibility of ad-hoc group formation.

Prior work has also suffered from a lack of precise control over the allocation and exercise of privileges within the TD. This lack of granularity means that sometimes nodes must be trusted with sensitive capabilities that might otherwise be witheld from them, lowering the overall security of the Trusted Domain. In both of the works referred to above, a node is either a member of the TD or not, and members may exercise any privilege granted to TD nodes. Our model provides granular control over privileges, breaking the TD up into a number of *levels* at which a node may participate. Each level contains a set of privileges which a node at that level may exercise. At higher levels, a greater investment of resources is required, and a greater set of privileges are allowed. Our model guards against masquerading by requiring nodes to commit greater resources at higher levels, and over time. Continued good behavior by fair and honest nodes is rewarded in this way. Bad behavior results in expulsion from the Trusted Domain. In colluding with

malicious nodes, well behaved nodes risk losing privileges gained over time; collusion is thereby discouraged.

In the following section, § 2, we discuss the rationale for the model upon which our work is based. Following that, in § 3, we discuss he implementation of the model as a Peer to Peer (P2P) file sharing network build on top of the Bouncer Toolkit, using the TS-RSA algorithm as the underlying threshold cryptographic primitive. Finally, in § 4 and § 5, we present the results of performance and scalability testing based on laboratory experimentation with our implementation.

Chapter 2

The Model

The model which we now present is intended as a solution to providing essential security properties and fairness in a peer group, while allowing ad-hoc formation and dealing with the problem of masquerading. Our model is founded on a number of essential assumptions which are roughly derived from observations of social behavior, and which we now discuss.

Consider two people, Alice and Bob, who have just met for the first time. Neither trusts the other person, but they both wish to accomplish some task that neither alone can complete. Let us assume that both see that they must unite to accomplish the task, and that they begrudgingly join forces to get the job done. Alice labors to complete her part of the task while Bob labors to complete his. If each periodically shows the other the work that they have completed, over time they will both come to trust that each is working hard for the collective good. As long as Alice and Bob decide on a suitable period, then the most undesirable thing Bob can do as far as Alice is concerned is stop working just after he shows Alice his work. This would mean Alice continued to labor for an entire additional period, while Bob stood by. Presumably, next time Alice saw Bob's work, she would realize he had been lazy, and would decide not to continue to work with him.

On the other hand, the longer Bob continues to work hard, the longer Alice can see that he shares the same goal as her, and is therefore likely to continue to work hard. Let us assume that it takes a bit of time for Alice to walk over to where Bob is working, observe that he has completed the work expected of him, and then walk back to where she is working and begin again. Over time, Alice will grow to trust that each time she makes a record of Bob's work, it will be as she expected.

Alice can propose to Bob that they increase the period at which they check in with each other. In doing so, she has admitted to some increased trust of Bob, given his continued good behavior, and has exercised this increased trust by way of increasing the check-in period. She can never be completely sure that Bob is not out to cheat her. But she grows increasingly more confident that Bob is honest and fair, because he has committed so much of his time and energy to completing their shared task. So the assumption is that because Bob has invested so heavily into completing the task, he will not jeopardize its completion by being lazy; rather, he will continue to work hard.

Eventually, Alice and Bob will complete what they set out to do. Before they part ways, Alice gives Bob a token that tells the rest of the world to what extent Alice approves of Bob. Later, if Bob runs across another person who wishes to join him in completing some other shared task, he can use the token to show them that he has already established a rapport with Alice.

Our model is predicated upon the observations above. We start by assuming that some core group of nodes trust each other mutually. Each new node that wishes to take part in the peer group is invited to do so as long as enough members approve; the first time a node joins a group members always approve. Over time, its behavior is observed, and offered increased privileges. With the increase in privileges comes an increase in expected participation. As more members are added to the group, the number of observers increases, each one adding their unique perspective. In total, their collective wisdom is

more accurate than the wisdom of a single individual, to the benefit of the group.

If at any time a node acts maliciously, unfairly, or fails to commit the resources required of it for participation, it is expelled from the peer group. Honest, fair nodes are discouraged from being malicious or colluding with malicious members because in doing so they risk losing their membership. Members receive tokens proving their membership and validating their participation. The model can be used iteratively, with one peer group requiring membership in a more junior peer group a criteria for joining. In this way, malicious or dishonest members may be filtered out as they proceed upward through a hierarchy of peer groups.

The social model presented above is suitable for situations in which some degree of misbehavior can be tolerated. Bob can always choose to misbehave at some level, and for however long it goes undetected, this behavior must be accepted. In a situation where the result of misbehavior is serious, the model can still be used, but over time, given increasingly more observers, the ability to detect and expel members is increased. Thus, the longer the model is used in such a situation, the greater its ability to deal with misbehavior.

2.1 Construction of the Initial Group

Our model begins with a number of peer nodes coming together to form a Trusted Domain, a process which we term *Initial Group Construction*. As stated in the previous section, the model must be used in a largely honest environment if it is to function properly. In particular, the number of colluding dishonest nodes should not be equal to or exceed the threshold t. If the Initial Group forms with a number of dishonest nodes at least equal to t, then the group is compromised from the start. The easiest way to form a TD with the desired properties is for a number of nodes with prior knowledge or out-of-

band experience with each other to form it. We note that this may not be possible in an entirely ad-hoc setting, however.

We present a technique for initial group formation that guarantees that group formation occurs as desired. From the Byzantine Generals Problem (BGP) [22], we know that given any $3m+1$ honest parties, even in the presence of m malicious parties we can form an initial group P_0 and defeat attempts by m nodes to disrupt its formation. Let us assume $m = 1$ malicious nodes, and therefore $3m + 1 = 4$ total nodes. Call these nodes $p_1...p_4$, and let p_n refer to any single $p \in \{p_0...p_4\}$. Let us assume that each node possesses a cryptographic certificate that is verifiable as belonging to the owner. Let us assume that all 4 nodes have such certificates, with public keys for these certificates known to all other nodes.

P_0 forms between two times, T_0 and T_1 ($T_0 < T_1$). Starting from time T_0, all p_n use a distributed threshold cryptographic algorithm to bootstrap a shared private key with threshold t. Again using a Byzantine Agreement Protocol, at least $3m = 3$ honest nodes posses shares of the shared private key at the conclusion of the exchange. Note that it is necessary to bound the threshold t between $m + 1 = 2$ and $3m = 3$ above, given $m = 1$ potentially malicious nodes. Failure to bound t in this way would allow for the creation of an unusable key, or a key which was insufficiently distributed amongst p_n such that m nodes could compromise the group secret key. We therefore assume honest behavior by at most $3m$ nodes during the exchange, and accommodate malicious behavior by at most m.

Between T_0 and T_1 it is assumed that no honest node becomes malicious, and all honest nodes participate in the exchange. The actual value of the interval between T_0 and T_1 is unimportant as long as the assumptions hold for that interval. The exchange could take place over seconds or days. Typically a short exchange provides little opportunity for attack, and so would therefore be a more suitable choice than a long interval. Once

T_1 elapses, at least $3m = 3$ members of p_n have formed the peer group which we now simply term P.

Recall that the group P has formed in order to collaborate to achieve some goal. The exact details of how they will collaborate is articulated in a Group Charter (GC), an `X.509` Certificate [19]. There are a number of ways in which all p_n can agree upon a GC. If the group is forming entirely ad-hoc, perhaps the easiest way is to select one of a number of predefined charters that might exist. Otherwise, a voting algorithm might be used. Once the GC is decided upon, it is partially signed by each p_n, and bound to the group public key. The GC contains at minimum the following elements.

Figure 2.1: Group Charter Elements

- An NxN **Policy Matrix** O $\begin{pmatrix} o_{0,renwal} & o_{0,upgrade} & \dots & o_{0,N} \\ o_{1,renwal} & o_{1,upgrade} & \dots & o_{1,N} \\ \dots & \dots & \dots & \dots \\ o_{N,renewal} & o_{N,upgrade} & \dots & o_{N,N} \end{pmatrix}$
- An NxN **Capabilities Matrix** A $\begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,N} \\ a_{1,0} & a_{1,1} & \dots & a_{1,N} \\ \dots & \dots & \dots & \dots \\ a_{N,0} & a_{N,1} & \dots & a_{N,N} \end{pmatrix}$
- The Threshold Cryptographic primitive to be used
- The initial threshold t

These values essentially define the group, and the requirements for members of the group. Each member in the group belongs to the group at some level l, binding it to a row in both A and O. Members of the Trusted Domain, such as all of those in P belong to the group at the highest possible level, the highest-numbered row of A and O. New members are accepted at the lowest level, $l = 0$. Each row of the Capabilities Matrix, corresponding to some level, contains a list of the actions which a group member may take at that level. Any actions that the group allows that are not listed at that level are prohibited for members at that level. A given row of the Policy Matrix lists the extent

to which members at that level *must* exercise some capabilities, and the extent to which members must limit their use of others. These matrices, and in particular the values of $o_{renewal}$ and $o_{upgrade}$, are discussed at length in the following section.

2.2 Admission Process

Given the Trusted Domain P, let us examine what happens when some new node n wishes to join the group. To begin with, n must have generated, as prerequisite, only a public/private key in order to join; there is no other requirement. First, n requests the Group Charter from any of the nodes in P. Assuming n finds the contents of the GC to its liking, it sends a GMC Request (GMC_{REQ}) signed with its private key to at least t members of P. The GMC_{REQ} contains the following.

Figure 2.2: GMC_{REQ} Attributes

- A ***Level*** $Level$
- A ***Start*** time $Start$.
- An ***Expiration*** time $Notafter$.

The GMC_{REQ} is an `X.509` Request [28] containing some of these values as `X.509v3` extensions [12], while others are included in the message in which the GMC_{REQ} is sent. The node n sets the value of $Level$ to 0, as this is its initial request to join the group. $Start$ is set to the Universal Coordinated (UTC) time. The value of $Notafter$ determines when the certificate expires. Each list of policy elements $o_l \in O$ contains one element that determines for how long a GMC at that level is valid. The value of $Notafter$ is simply calculated by adding this element to $Start$ to begin with.

This is the first point at which admission is controlled by way of using t as a proxy. If no t members find n acceptable as a new member, it cannot obtain membership. Members of P would consider any prior knowledge of n that they might have, as well as checking

the values sent by n in its GMC_{REQ} . Recall that the number of malicious nodes is less than t for P making it impossible for a malicious coalition in P to approve members arbitrarily. If fewer than t members decline to admit n, membership is denied. Having declared n admissible, at least t members of P sign a Group Membership Certificate (GMC) for n constructed from the GMC_{REQ} and using the same values.

Two elements of the Policy Matrix O are required to be defined for each Group Charter at all levels. For some level l the element $o_{l,renwal}$ defines a renewal time in seconds that is added to the current time, when each GMC is issued, to arrive at an expiration time for each GMC. The element $o_{l,upgrade}$ marks the minimum time that a node must have participated in the group before it will be considered for membership at level l.

As the last section mentioned, the level l is used to refer to a set of policy elements o_l, and a set of capabilities elements a_l to which n is to be held until the certificate expires at $Notafter$. Each of these matrices specifies the way in which all honest nodes in the group will allow other nodes to make use of their resources. If n requested some resource and the capabilities matrix for their level did not allow access, any honest node receiving this request would reject it. Or, for example, if the policy matrix specified that up to a particular number of messages of some type may be sent, and n sent more than the allowed number to some host y, then y would refuse future messages from n.

While its GMC is valid, n must only attempt to make use of the capabilities granted to it by a_l. Should n try exceed its capabilities, as we describe above, any node receiving such a message from n would fail the message, record the event, and refuse further messages from n. The same is true for elements in the policy matrix. On the other hand, should n fail to meet the requirements of the policy matrix by not sending *enough* of some message, the result would be slightly different; this outcome is addressed shortly. In all cases, the recipient of some message can be sure that some message originated from n because such a message would be signed by n using its private key.

Before $Notafter$ elapses and its certificate expires, n must apply for a new GMC. This process is essentially the same as the initial application with a few differences. The value $Notafter$ is again arrived at by taking the current time and adding the expiration time from the policy matrix to it. Again, the GMC_{REQ} is constructed, and submitted to each of the (at least) t nodes $p_i \in P$ with whom n has interacted. Each p_i evaluates n, checking to see that it has exceeded no $a_i \in A$, has met all $o_i \in O$ and has not exceeded any $o_i \in O$. So long as this is true, all honest nodes $p_i \in P$ partially sign 1.1 the GMC_{REQ} submitted by n. As we can see here, n is thus required to behave in accordance with both matrices with at least t other nodes. In this way, t is used as a proxy, forcing n to interact, and do so honestly.

If n has violated either the capabilities or policy matrix for some honest p_i, this node would refuse to sign, and n would be unable to acquire its new GMC. Therefore it is in the advantage of n to interact with, and request its GMC_{REQ} of more than t nodes in order to tolerate some misbehavior during GMC renewal. Specifically, if n can interact with $2t - 1$ nodes, it can guarantee successful renewal[1].

After n has participated at level l for the time period $o_{l+1,upgrade}$, it may make a GMC_{REQ} of P for membership at a higher level. So long as, just as above, n has participated in accordance with the matrices, its request would be approved. For each increase in l, n will be granted an increasing subset of capabilities from A, and held to a stricter and more demanding set of policy elements from O. By way of controlling the progression of new members through the various levels of A and O, members of P are able to control the extent of the trust they give to new members. Over time, after participating for $o_{N,upgrade}$, n could request and be approved for the maximal level l, effectively joining P as a full member. Once approved, n may request a share of the group secret key S. A share of S is calculated for n in accordance with policy, and the signature scheme used,

[1]Recall the assumption that no more than $t - 1$ malicious nodes exist in the chosen environment

and is securely dealt to n.

2.3 Iterative Use of the Model

While the model has been designed to function in an entirely ad-hoc manner, this is not necessarily where it can be most powerfully applied. Consider two peer groups P_x and P_y where entrance (membership at level 0) into P_y requires membership in P_x at the highest level (l_N). Some node n at l_N in P_x applying for membership to P_y has been observed to be honest for some extended period of time by the Trusted Domain of P_x. Given that, the t nodes of P_y considering n for entrance at level 0 may have a considerably higher basis for accepting n at level 0 than the t nodes of P_x that presumably had no prior knowledge of n.

By iteratively requiring membership in other groups in this way, a hierarchy of peer groups can be built with groups at the top of the hierarchy having increasing confidence in the honesty of new nodes. This is an intentional quality of kour model that once again follows the social trust model.

2.4 P2P File Sharing Example

As an example of how the model might be used, consider a Peer to Peer file sharing group G collaborating to exchange files, as nodes in a Gnutella [8] file sharing network might. First, we construct a Group Charter to represent the aims of G. Recall from Figure 2.1 that a GC for G will be composed of the following elements.

Figure 2.3: Group Charter Elements for G

- An NxN **Policy Matrix** O $\begin{pmatrix} o_{0,renwal} & o_{0,upgrade} & \dots & o_{0,N} \\ o_{1,renwal} & o_{1,upgrade} & \dots & o_{1,N} \\ \dots & \dots & \dots & \dots \\ o_{N,renewal} & o_{N,upgrade} & \dots & o_{N,N} \end{pmatrix}$
- An NxN **Capabilities Matrix** A $\begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,N} \\ a_{1,0} & a_{1,1} & \dots & a_{1,N} \\ \dots & \dots & \dots & \dots \\ a_{N,0} & a_{N,1} & \dots & a_{N,N} \end{pmatrix}$
- The Threshold Cryptographic primitive to be used
- The initial threshold t

Let us say that the group G allows for three capabilities: *send*, *receive*, and *search*. Building the Capabilities Matrix for G based on these possibilities we have:

Level	Send	Receive	Search
1	X		
2	X		X
3	X		X
4	X	X	X

Table 2.1: Capabilities Matrix for G

For each level l, an *X* marked for the capability in each column indicates that the capability given for that column is available. Capabilities with no *X* are unavailable for that level. At the first level, looking at Table 2.1, nodes may only send. The next two levels allow nodes to search as well, and finally the last level allows nodes to receive files.

Let us say that the policy matrix for G is as follows.

level	Bytes	Searches	Renewal	Membership
0	$Bytes_0$		$Renewal_0$ sec	
1	$Bytes_1$	$\leq Searches_1$/min	$Renewal_1$ sec	$Membership_1$ sec
2	$Bytes_2$	$\leq Searches_2$/min	$Renewal_2$ sec	$Membership_2$ sec
3	$Bytes_3$	$\leq Searches_3$/min	$Renewal_3$ sec	$Membership_3$ sec

Table 2.2: Policy Matrix for G

Exploring Table 2.2, we see that first of all each of the 4 levels contains 4 policy elements. Recall that for all levels l, all policy elements must be met for certification at that level. Each $Bytes_l$ refers to the minimum number of bytes that must be sent from some node n to each Trusted Domain host from whom a GMC will be requested; recall that this is at minimum t such hosts. For each increase in l, some node n is required to transfer an increasing number of bytes to each host. Note that meanwhile, n can do nothing but send files and search. The policy elements at level 0 of the matrix, the first row, can be considered the *entrance fee* that a node n must put forward to begin participation in the group. It is the minimum required policy with with n must comply in order to participate. At minimum, any node wishing to join G must consider these elements with respect to the value of t after receiving the GC for the group. Initially, in this example, n can do little by way of misbehavior. Eventually, at the highest level, n may join the Trusted Domain and also receive files.

As required by Figure 2.3, each row of the policy matrix includes the two variables $o_{l,renewal}$ and $o_{l,upgrade}$ which specify the time, in seconds, that a certificate is valid before it must be renewed, and the time, in seconds, that a node must participate in the group before it may upgrade to some level l. The variable $Renewal_l$ specifies the renewal period, and $Membership_l$ specifies the time before upgrade may occur. By increasing or decreasing each $Membership_l$ a group may require a very long or very short observation

period at each level. The longer $Membership_l$ is, the greater confidence members of G may have that some node n will continue to behave correctly[2].

In combination with $Bytes_l$ each $Renewal_l$ implies a rate of transfer that some node n must meet. If $Bytes_1$ were set to 100 bytes for example, and $Renewal_1$ were set to 100 seconds, then some node n at level l must transfer at least 100 bytes in the renewal period of 100 seconds, for a rate of at least $100/100 = 1$ byte/sec. Note that n must renew its certificate *before* expiration, and so would want to send at a rate slightly higher in order to allow for some time to solicit for and acquire its new certificate.

Consider the selection of t for this example. With four nodes in G, selecting a t of 3 ensures that the greatest number of nodes in G will observe some new member n, while still tolerating a single malicious node. But, the failure or compromise of any other node in G renders the shared private key unusable by G. Selecting a t of 2 allows for tolerance of a single fault or compromise in G. Selecting t as 4 renders the group G incapable of tolerating any faults or failures. On the other hand, selecting a t of 1 allows maximal tolerance, but clearly invalidates the use of the threshold, giving all nodes in G the full authority to sign as G.

In Section 2.5 we discuss all manner of attacks against the model. Here, we consider one attack against this example. If an attacker n is able to masquerade as t hosts, it can participate at the highest level in G and acquire t shares of the shared private key, compromising it and allowing n to sign as G arbitrarily. If the values of $Bytes_l$ and $Renewal_l$ have been set high enough, it may be possible for the sustained rate in bytes per second required to masquerade as t nodes to be made to exceed the physical capacity of n.

Let n be connected to G through some physical connection with a maximum possible rate of 150bytes/second. Let us say that $Bytes_3$ and $Renewal_3$ are set such that partici-

[2]See § 2

pation at level 3 requires a sustained rate of 100bytes/second. In this case, n will not be able to satisfy this rate more than once, and so will succeed in disclosing more than one share of the group private key in this way.

In some cases, unique, observable properties of n may assist in guarding against masquerading. Binding IP address to the GMC for each node, for example, might make masquerading more difficult. In some settings, a SIM id[3] might exist, which is a much stronger property that can be used for this purpose. The more unique, and difficult to forge the property, the stronger it is in in guarding against masquerading. Lacking such an identifier, the choices made for the policy matrix can still assist in preventing masquerading. Care must be taken, however, as setting the policy matrix requirements too high would limit the types of hosts that could participate in the group.

2.5 Attacks Against the Model

Now we consider a number of attacks against the model. The behavior of any group member can be modeled given the level at which it is held to the capabilities matrix A and policy matrix O. Given some l, we know what the member can do, we know to what extent it can do it, and we know for what length of time. Since the highest level affords the greatest set of capabilities from A, let us assume that this is where the member can do the most damage. We will start our analysis by examining two distinct types of possible attacker behaviors.

We categorize the types of attacks brought against the model into *greedy* and *malicious*. A *greedy* member wishes to do less than is required of it while benefiting as much as possible from its membership. A *malicious* member wishes to exceed its privileges, or use its privileges to the disadvantage of the group. So, a greedy attacker will violate

[3]Such as can be found in most cell phones, for example.

some part of the policy matrix O, while a malicious attacker will exceed or abuse its capabilities from A. Later, we consider an arbitrary attacker that can exhibit these and other behaviors.

2.5.1 Greedy Members

Let us consider an attacker $attacker_g$ modeled under our first example, a group of file sharing nodes. In this example, the maximum $a_l \in A$ and $o_l \in O$ allow $attacker_g$ to conduct searches and receive files, while requiring $attacker_g$ to share files at some rate. If $attacker_g$ is greedy, it might try to share files at lower than the specified rate, or search more than the is allowed. Conversely, $attacker_g$ might attempt to share greater than the required $Bytes_l$ or search fewer than the maximum allowed number of times. This behavior benefits the group, and does not benefit the attacker, and is therefore not considered greedy. Any attempt by $attacker_g$ to search beyond what is allowed will be rejected by any honest member of G, and future requests from $attacker_g$ will be ignored.

On the other hand, nodes to whom $attacker_g$ is sending files can not know that $attacker_g$ is being greedy by sending less than the required $Bytes_l$ until its certificate expires; until that time $attacker_g$ might still satisfy the requirement. Eventually, its certificate becomes invalid, or $attacker_g$ makes a GMC request without having met this requirement, and all honest nodes refuse any GMC request from $attacker_g$ on the basis of this violation of policy.

In both cases of greedy behavior above, the behavior is tolerated for at most $Renewal_l$. There is a mechanism by which this time period can be lowered. If G makes use of an `X.509` Certificate Revocation List (CRL) [19], t honest nodes who have detected misbehavior by $attacker_g$ may add its GMC and public key to the CRL. Once the CRL is propagated, all nodes in G will refuse requests from $attacker_g$. In this case, greedy behav-

ior would be tolerated only so long as no t members of G had detected such misbehavior, and once detected, only so long as required for the CRL to propagate.

2.5.2 Malicious Members

Now we consider a malicious attacker $attacker_m$ for the same example. Such an attacker could try to exceed its capabilities by receiving files or searching before these privileges were allowed. Just as in the case of a greedy attacker, examination of the GMC for $attacker_m$ would show that these capabilities were not allowed for its level l, and attempts to make use of these capabilities would be denied, and future requests from $attacker_m$ ignored. Again, if a CRL were used, expulsion of $attacker_m$ could occur as immediately as noted above.

Now, consider the danger that compromise of the group shared secret poses. A malicious attacker might try to masquerade as, compromise, or collude with a sufficient number of other nodes in order to acquire the group secret key. Recall that t total nodes must act to compromise of the group shared secret. In our model we assume that no such t nodes ever exist.

As a rationale for this claim, recall from § 2.4 that masquerading will not succeed in some environments; if, for example, the group consisted of IP-network nodes, and each attacker was only able to make use of a single network address. Or if the group consisted of nodes with observably unique transmission signatures. More generally, any environment where the attacker cannot forge or acquire more than one unique identifier. Furthermore, recall that even in the absence of such an identifier, the policy matrix may be constructed in such a way as to preclude masquerading.

2.5.3 Collusion

It is not by accident that our solution to the problem of masquerading leaves the attacker with no choice but to collude with other members. An observation in the development of the model was that if we can force collusion on the attacker, and at the same time make collusion very unattractive to honest members, the attacker will be thwarted. The means by which we provide other members with an incentive not to collude is twofold.

First, by increasing their privileges over time, and making good policy choices, we can reward well behaved members while not taxing them too severely for their good behavior. Second, by forcing continued good behavior over time, we build each members' commitment of resources over time. The idea is that after so much resource commitment on the part of each honest group member, the rewards afforded by compromise of the group secret is not worth the risk of losing membership status. This is true as long as the rewards for attaining higher levels outweigh the benefits of misbehaving at these levels. This makes choices in the policy and capabilities matrix crucial in order to strike the appropriate balance.

Chapter 3

Experimental Design

Above, we suggest a model that we claim can provide security properties and fairness in a peer group. We hypothesize that the model will provide these properties without significant overhead, lending it to use in peer groups with substantial thresholds and membership counts. We propose to implement the model described in § 2.4 as the basis for the creation of security mechanisms in a Peer to Peer (P2P) file sharing network. In a later section, we test the implementation, observing its performance, and confirming correctness of operation.

Our implementation is built upon the Bouncer Toolkit [1] version `0.7` [3] and constitutes a contribution of greater than 15,000 lines of `C` code. We begin by discussing the toolkit, and then proceed to discuss our specific contributions.

3.1 The Bouncer Toolkit

The Bouncer Toolkit[1] is a framework for peer group admission control written in C by a research team lead by Dr. G. Tsudik [6] at the Secure Network and Computing Center at UC Irvine. Bouncer is essentially an implementation of the ideas discussed in [27]. It provides support for the elements seen in Figure 3.1.

Figure 3.1: Bouncer Toolkit Provisions

- Creation of a Group Charter
- Static and Proactive secret sharing primitives and other signature schemes
- Admission control (admission, expulsion) using the selected signature scheme

Included in the 0.7 release of the toolkit is support for the following signature schemes.

Figure 3.2: Bouncer Toolkit Signature Schemes

- PS: Plain Signatures
- TS-RSA: A Threshold-shared implementation of the RSA algorithm
- TS-DSA: A Threshold-shared implementation of the DSA algorithm
- ASM: Accountable Subgroup Multisignatures [23]
- GS: Group Signatures

Each of the signature schemes listed in Figure 3.2 is built as a static C library. Bouncer uses OpenSSL version 0.9.6g [9] which implements the necessary underlying cryptographic primitives such as RSA and DSA and provides the necessary arbitrary precision math library[2]. OpenSSL also provides Bouncer with support for the X.509 ASN.1-like [19] syntax, and ASN.1 [14] Distinguished Encoding Rules (DER) [15] format encoding and decoding. Bouncer includes a packet handling library which implements any exchanges over IP which may be needed by the signature schemes.

[1] All comments we make in this section that referencing source line numbers of this toolkit shall apply to *our* source release packages rather than the original toolkit.

[2] This exact version of the toolkit must be used as versions of OpenSSL are not guaranteed to be binary compatible from one release to the next.

Furthermore, Bouncer includes a test suite that makes use of each of the signature schemes and performs admission control for some test group. A setup script generates the test group to be run. Two daemon processes are required to run a test group. The *Group Authority* Daemon (GA) and the *Certification Authority* Daemon (CA). The CA is used only for test purposes to sign the Group Charter and perform some other limited functions; ultimately it is not required by the implementation. The GA, on the other hand, is required, and it is important to discuss its role as its use fixes the infrastructure and creates a single point of failure.

The use of the GA is primarily due to the need to accurately keep track of the membership for some peer group [25]. The problem is a difficult one, and is compounded by the fully asynchronous, decentralized group setting. The use of the GA ultimately needs to be deprecated in favor of a better distributed solution analogous to that used by, for example, Gnutella [8]. Nonetheless, when the GA is run, it prompts the user for various parameters used to create a Group Charter. Once the Charter is created and signed by the CA, an automated process, the GA becomes the membership authority mentioned above.

The most important component of the Bouncer test suite is the application server, called `apps`. Bouncer is designed to run in test mode on a single system or across a number of networked hosts. When running across a network, the setup script must be made aware of the address of each host that will participate in the group, as there is no discovery mechanism. If the setup script is instructed to create a test mode group, each host executes on the same machine, and each listens for incoming packets on a base port plus some offset.

Upon loading, each apps instance contacts the GA downloads the Group Charter, and is told the current membership count. If the count is above the threshold t selected in the GC, each member requests a Group Membership Certificate (GMC) from its peers, each of which then then performs admission per the selected signature scheme. If the count is

below the threshold t, the host contacts the GA which performs a bootstrap for the host, with the same result as above[3].

Once loaded, the apps do not do very much in this test suite. Their purpose is to serve as application servers, responding to whatever messages might be sent to them by other hosts making use of the selected signature scheme. Though limited, their behavior serves its purpose, which is to exercise the full range of capabilities of the Bouncer Toolkit.

3.2 Our Implementation

First, we broadly outline the behavior of our implementation, and following that we will detail the work that was done for each significant change to behavior and functionality of the system.

We selected the Bouncer toolkit for the basis of our implementation after having worked with the Secure Gnutella [7] sources briefly. We found that the SGnut sources were themselves too immature, and were based on an earlier version of Bouncer (`0.5` [2])[4]. Bouncer `0.7` appeared to be the more mature, stable choice.

3.2.1 Changes to Setup and Installation Process

The first change in our implementation was a rewrite of the original installation and setup process. Bouncer `0.7` expected to be installed to the source directory; we rewrote the installation process to allow for installation to any directory which made deployment and development much easier[5]. Once installed, two scripts created a series of test directories

[3]This statement is limited to all Threshold signature schemes, which we limit our interest to.

[4]A race condition exists in version `0.1.2` of the SGnut sources which prevents them from working properly. In `cli_interface.c:175` a thread is started with the entry point `gad_main` which binds to and listens on `GAD_TCP_PORT`. Shortly afterward, `GAD_TCP_PORT` is connected to by the original process thread; but because no coordination is used, there is no guarantee that `gad_main` is listening on this socket yet.

[5]The `setup` and `certgen` programs require certain files from the source directory to be present at runtime which do not get copied to the installation directory.

from which the test suite could be run. Both were implemented as `C` programs that outputted shell script to a file, and made a `system` call to execute the resulting shell script. Furthermore, prompts and output in these scripts was confusing, and generally, the scripts were fairly fragile. We consolidated them into a single, robust shell script.

3.2.2 Broad Outline of Our Implementation

Once installed and configured using our setup script, the implementation functions as follows. First, the CA and GA are started. In our implementation, all Group Charter parameters are specified in the setup process, rather than at GA run time. Then, between the time periods T_0 and T_1, n hosts form the Trusted Domain as follows[6]. First, each contacts the GA and downloads the Group Charter. Then, each constructs a GMC Request and submits it to the GA, which checks and approves the request. The GA sends the host its GMC, and a share of the group secret. Following this, each host waits until T_1 has elapsed, at which point the TD has formed.

Each host, which we will refer to hereafter as a *servant*, is composed of three processes, without which no servant is complete. One subprocess handles all incoming packets, processing and responding to them. A second implements the honest behavior expected of nodes participating in the group. The third is responsible for occasionally sending a *Ping* message to the GA. A Ping is sent after every GMC acquisition so that the GA always knows which valid nodes are online, and can inform, or respond to requests to inform, other servants of the current domain topology.

With the Trusted Domain formed, other servants may join. Each servant in the network must send as much data as required of it by the policy matrix. Honest servants will transmit to as many hosts as their bandwidth allows, and each host has a configured max-

[6] T_0, T_1 and n are specified during setup

imum bandwidth. Any changes in domain topology are transmitted to hosts by the GA. Any malicious or greedy hosts are blacklisted by each servant when detected.

With the general outline of the behavior above in mind, we now proceed to detail the changes we made during implementation.

3.2.3 Group Charter Extensions

The Group Charter is an `X.509` certificate signed by the CA, containing the following parameters as `X.509v3` extensions.

Figure 3.3: Group Charter Parameters

- Group Name
- Signature Type
- Threshold Type (Static, Dynamic)
- Dynamic Threshold (as a Percent of Membership Count)
- Static Threshold
- Below Threshold Policy (use GA or use all current members)
- Dealer Assignment (GA or Group)
- Policy Matrix
- Capabilities Matrix
- Model Type
- Format of Matrices

We now examine some of the extensions from Figure 3.3. A *Below Threshold Policy* may be set only for non-Threshold signature schemes allowing nodes using those schemes to decide what to do when membership falls below the threshold. Since we are using a Threshold signature scheme only in our model and implementation, this is ignored. Furthermore, nodes always make use of the GA when the Trusted Domain is forming regardless of whether membership count is below threshold, and afterward, it is never below threshold. Should sufficient members leave causing the count to fall below, the

group dissolves. In our implementation we chose to use a Static Threshold for testing purposes, therefore the value of the extension *Dynamic Threshold* is also ignored.

The *Dealer Assignment* extension indicates whether the group, or a trusted dealer, is to bootstrap the shared secret. We take this opportunity to note, importantly, that in our implementation, we use TS-RSA only, and group bootstrapping is available only to groups using TS-DSA. Initially we chose TS-DSA for this reason, but we discovered that in Bouncer `0.7`, for what seems to be testing purposes, the TS-DSA implementation performs a number of operations wherein ultimately the secret is reconstituted at a single servant[7]. Using TS-DSA would have meant changing this, and as we discuss in § 3.2.7, some rewriting of whatever threshold primitive was chosen was known to be required. It was therefore decided that we make use of the most viable, and only, remaining threshold primitive which provided a decidedly better starting point.

3.2.4 Implementing the Matrices

Our first change to the Bouncer toolkit was the addition of the last four extensions seen in Figure 3.3 to the Group Charter. First, this required writing the necessary OpenSSL callbacks, extending the existing code, and choosing appropriate `X.509v3` Object Identifiers (OIDs) [11]. The next step was to implement code to pull the matrices out of the GC where they are `ASN.1` DER encoded, and make them presentable to the rest of the servant processes.

Two extensions from Figure 3.3 determine how the matrices are to be encoded and decoded. First, the Group Charter extension entitled *Format of Matrices* determines how the matrices are encoded. In our example we encoded each matrix as ASCII formatted text using the pipe character to delineate column boundaries, and the space character

[7] See `apps/main.c:542` which calls `GAC_Secret_Recovery` and "computes $S = \sum ss(i)lag_i(0) \pmod p$" (`libgac/gacLib.c:210`)

to delineate row boundaries. Our implementation is flexible, allowing for the matrices to be encoded arbitrarily, for example as XML [16] or in a proprietary binary format. The encoding used is separated from the format for the content of the matrices, which is described in the extension entitled *Model Type*.

The combination of these two extensions determines how the matrices are parsed out of the Group Charter and presented to the application, once they are converted into OpenSSL native format from their DER encoding. Callbacks are used making the addition of additional encoding/decoding types trivial. A handler examines the Group Charter, and calls the appropriate decoder based on the encoding of the matrices; in this case only one encoder is defined for the format described above. Once decoded, the data are passed to the decoder specified by *Model Type*. Again, only one type and handler are defined. The default decoder turns the matrix into a two-dimensional array of long integers.

3.2.5 GMC Extensions

Figure 3.4: GMC Fields

- Expiration
- Level
- Start

Similar to the extensions we made to the GC, a number of Group Membership Certificate (GMC) extensions needed to be created. While similar to the GC changes we made earlier, in this case the GMC used no existing `X.509v3` extensions. In particular, we were concerned with integrating three values into the GMC, each of which can be seen in Figure 3.4. One of these values was the expiration time of the certificate. Each `X.509` certificate includes this parameter as part of the validity field [19], which meant that only two parameters needed to be added to the certificate as an `X.509v3` extensions. The

Level parameter is the level to which a servant is presently bound. The *Start* parameter is the UTC time at which the node was issued its first GMC, and can be used to determine eligibility for upgrading to higher levels.

We added the necessary functionality to enable these extensions to be added to both GMC Requests (which are `X.509` Requests [28]) and GMCs. A set of helper routines was created to decode these values from the GMC and present them to the application in a native format.

3.2.6 Changes to the Bootstrap Process

At runtime, each servant loads the duration of $T_0 \rightarrow T_1$, and the count of the number of nodes set to participate in the TD, each of which is available in a configuration file. We changed Bouncer such that two bootstrap processes are used, one for TD nodes (before T_1 expires), and one for non-TD (after T_1). If the time period $T_0 \rightarrow T_1$ has not yet elapsed, and the membership count is below the configured number of Trusted Domain hosts, nodes request a certificate and share from the GA as they are expected to become TD nodes. If T_1 has elapsed, they make use of a second distributed bootstrap process wherein t nodes are contacted for a GMC, and participate as non-TD nodes.

Bouncer has more than one bootstrap process as well, but there are significant differences between our implementation and theirs. Bouncer is designed to bootstrap up to t nodes, and make use of a below-threshold policy. In our implementation, below threshold always means that the TD is forming. We make use of a bootstrap period $T_0 \rightarrow T_1$, in order to guarantee that the Trusted Domain forms as expected. As discussed earlier, when the TD is formed, the membership count always remains above the threshold unless the group is dissolving. Importantly, in their implementation of the the TS-RSA protocol, a share of the group secret is always disseminated along with the GMC. It was therefore important to alter the protocol to suite our implementation.

3.2.7 TS-RSA Changes

We changed the TS-RSA protocol implementation in Bouncer so that GMC requests and Share requests became two different message exchanges. This was an important and necessary change because in our model, only certain nodes are privileged enough to be trusted with shares of the secret. Furthermore, while shares were disseminated in Bouncer to all nodes requesting a GMC, in our model only Trusted Domain (highest-level) nodes are granted a share of the secret. Obviously some kind of mechanism was necessary to prevent arbitrary disclosure of secret shares. We implemented this functionality in two parts, which we now discuss.

3.2.8 Monitor and Tracker

The Monitor and Tracker modules are our means of controlling access to all group resources in compliance with the policy and capabilities matrices. The Monitor is composed of essentially two parts. First, a set of helper functions check various values against the matrices. Second, a set of routines exist that are able to determine, from a number of different factors, whether any given request or operation should succeed. A node must request a bootstrap, for example, only when the interval $T_0 \rightarrow T_1$ has not elapsed. When a request to bootstrap is made of the GA, it is the Monitor that ensures that it is the correct time, and that the number of Trusted Domain nodes is lower than the number that the setup script configured.

The Tracker module, on the other hand, keeps a log of packets received from all hosts in the network that are of concern. In particular, each occurance of a *send*, *receive*, or *search* message is tracked, as well as how much data was sent, and so forth. The Monitor is also aware of all Tracker data, so that decisions can be made that affect policy such as how many messages of some type may be sent for each GMC duration.

Recall that packets are processed by a subprocess dedicated to this task. All packets logged by the Tracker are handled by a child process spawned by this subprocess. It is therefore necessary to use some inter-process communication mechanism to share this data between the components of each servant. We chose to use shared memory segments and semaphores to affect this. The Monitor is a simpler module and does not need to record any data with one exception. When a monitor check fails, in many cases a host will be Blacklisted. Blacklisting a host means all future interactions with that host are negated. It was necessary to affect a number of changes before blacklisting could be implemented.

3.2.9 Networking Changes and Blacklisting

Our implementation of Blacklisting functionality required a number of significant changes to the Bouncer toolkit network layer. First, the implementation was highly fault intolerant. Any failure to send or receive from a network host caused the application to exit with an error message. This was undesirable, but it quickly became obvious how insecure this problem really was. Taking down any single host caused the entire network to fail due to a domino effect where one host went down and others followed as each was unable to send to some other node. Furthermore, packets were sent from a parent process and blocked waiting for a response from the recipient, causing whatever the parent was doing before blocking to be delayed indefinitely.

We added significant fault tolerance to the network infrastructure in the Bouncer toolkit. By default, any failure to send or receive from a host simply results in that host being blacklisted. Every blacklisted node is removed from the list of hosts that are to be interacted with. The Monitor module denies any request of any kind from a blacklisted host. Each entry on the blacklist is accompanied by a reason, and a timestamp. At a

number of important points, packets were also made to be sent from a subprocess, so that blocking was no longer an issue.

We further improved the network layer by making substantial changes to the packet layer. Initially, in the Bouncer `0.7` implementation, each node was essentially numbered, and could only rely on nodes less than its number being present in the network, as nodes were intended to be run in increasing order. This number was the offset, if testing locally, from a known base port. In networked mode, recall that each node was aware of the address of all other nodes in the network. Under this design, the GA would act as a rudimentary discovery mechanism by telling each host the membership count, which implied the hosts that were active (all hosts up to the membership count).

First, we replaced the test mechanism which used a base port and offset enabling testing on a single system. Using the IP aliasing capabilities of Linux, the test mechanism was made to work as follows. Each node on the local system uses the loopback network device and binds to a network address of `127.0.0.`x where x is the offset it would have used initially. Each host listens on the appropriate address, and when sending to other hosts, each was made to bind to the correct address.

This may sound simple, but it was an important change for the following reason. When using testing mode in the Bouncer toolkit, packets received from another node could not be attributed to that host easily. While each packet arrived on a uniquely different port, they were sent from random ports, and all packets were sent from the same IP address, the loopback address of `127.0.0.1`. With this change implemented, not only could each packet be uniquely attributed to a particular sender by IP address, but using the suite in testing mode became no different from networked mode; each mode simply tested a group of network hosts.

The discovery mechanism in Bouncer `0.7` was the very simple implied mechanism described above. Each host was aware of the membership count and its own number, and

this implied which other hosts it should expect to be up. We replaced this mechanism with a Ping process at each servant that sent a heartbeat to the GA each time a GMC was acquired. The GA was made to be aware of the topology of the network at all times. Any changes in the topology, as hosts renewed their certificates, or certificates expired, were propagated to all hosts in the network in a subprocess.

Once a packet is received by the network layer, it is decoded into an internal packet format and passed to a handler. Once in internal format, the packet did not contain any information about the sender except what might have been included for some particular packet type, for example a GMC. The IP address of the sender in particular was lost. We augmented the packet infrastructure to support some additional information, especially the IP address or hostname of the sender.

These changes were necessary for the Blacklist functionality to be propagated at all levels of the system. At a low level, during a send or receive operation to or from a host, our Blacklist functionality had already been implemented because at that level the IP was known. At a higher level the protocols that operated at the per-packet level were now able to use the same functionality, which was then implemented. Finally, packets in the Bouncer toolkit were always sent by way of a handshake involving a packet sent, and one received. We changed this so that packets could be sent one-way where no response was intended.

Signal handling in the Bouncer toolkit was also significantly improved, in particular as it affected the network layer. Initially, the network handling functions did not tolerate being interrupted due to receipt of an IPC signal, treating this as failure. We improved signal handling, enabling the network layer, and other similar functionality, to tolerate interrupts due to receipt of signals appropriately.

Finally, the toolkit originally used a single handler for all three network processes, the CA, GA, and servant. This was inappropriate for several reasons. First, the size and

complexity of each of these programs was increased as all available functionality was built in to them as the handler was built to handle *any* incoming packet which might be handled by any of the signature schemes implemented by the many libraries. Second, some packets were only ever meant for one of the three processes.

Some packets are only ever exchanged between the GA and CA, and others, in particular *P2P* packets discussed in § 3.2.10, are only transmitted between servants. But because of the use of a single handler, any process might receive any kind of packet, while only really being able to handle a particular subset. Receiving packets that a process could not *really* respond to would cause the process to make an attempt, usually ending in a crash. We broke the handler apart into three separate ones, with each process handling only packets which were appropriate for that process.

We conclude discussion of the networking changes by pointing out one change that was observed as an eventual necessity, but did not make due to its enormity in both time and complexity. The network layer infrastructure converts packets from internal to on-the-wire format, and transmits this data. The same infrastructure does the converse, converting received data back to internal packet format. As implemented this process is highly volatile, is non-portable both between system of differing endianness, as well as between processes compiled with differing alignment and padding of certain structures, and is consequently insecure[8]. For testing purposes, the implementation suffices, but crucially, this functionality would need to be rethought.

3.2.10 P2P Protocol

One of the biggest portions of the implementation was the construction of a new protocol simply called *P2P* which implemented *search*, *send*, *receive*, and other necessary message exchanges for our application. Each of the messages is accompanied by the GMC of the

[8] See `GAC_BuildPacket` from `libgac/gacSocket.c:187`.

sender, and upon receipt, each packet contains the address of the sender as well. Any message received can therefore be checked to ensure that the sender is authentic, using its GMC, and any misbehavior is logged and the host can be blacklisted. Our prototype network is designed to send fake data rather than actual files for simplicity, but the data is of the appropriate length with packets being the actual size they would be in a practical setting.

Our packet and protocol system is built in line with that of the Bouncer toolkit. Packets are constructed in an internal format with variables stored as `ASN.1` types. Before being transmitted, packets are serialized and written out to a buffer in DER format. When packets are received, they are tracked, and checked through the Tracker and Monitor modules.

3.2.11 Changes to Certification Process

In the Bouncer implementation, when the membership count is above the threshold t, nodes request their GMC from other members of the peer group. This process only occurs once, and only the nodes numbered 1 through t are solicited during this process[9]. We note that this departs from literature[10], but is otherwise a satisfactory starting point for our implementation.

To begin with, we altered the GMC request process so that it could occur any number of times. Then, we re-wrote the portions of this process that relied on hosts $1 \rightarrow t$ to allow it to make use of a list of any t hosts in the network based soley on their network addresses. Next, we separated out the TS-RSA share acquisition process, which had previously occurred in these same messages exchanges(see § 3.2.7). We built this share acquisition functionality into a similar but separate function.

[9] See `apps/main.c:393` in the original `0.7` sources.
[10] §6.2 of [25], point 3.

Apart from these changes, the functionality of each of these processes, share and certificate acquisition, take place much as they did in the original toolkit. First, a GMC Request is constructed in the form of an `X.509` Request [28]. Next, t hosts are solicited to ask whether they will sign our certificate, which is sent with the message. Once that is complete, a list of these hosts is constructed, each of their Distinguished Names is placed on the list, and the list is submitted to each of the solicited hosts. Each replies with a partial signature on the GMC, and these are sent to the GA which responds with our new GMC.

Crucially, the process above departs from literature in two places. First, as suggested by the model, and [25], when soliciting some GMC request, GMC_REQ_{new}, a node n expects to receive signatures on its new GMC, GMC_{new} derived from its request. In fact, a separate, static message transmitted alongside GMC_REQ_{new} is signed in the toolkit[11]. This is insecure, vulnerable to replay attacks for one thing, but is nonetheless the mechanism employed by this version of the toolkit. Doubtless this would need to be fixed in a future version. We chose not to fix this as it is trivially a proxy for what we are interested in, and ultimately it would not affect the results we were looking for.

Second, literature [25], and our model, suggests that the recipient is to recombine the partial signatures into the necessary full signature signed by the group private key. The use of the GA above is detrimental to distributed aims, but is nonetheless how this process works in practice in the toolkit[12]. Seemingly this was done for testing or simplification purposes.

This shortcut, however, was painful to discover, and ultimately use of the GA here should be superseded by the functionality described in literature. As the hard pieces of this process, specifically reassembly of the partial signatures, already occurs at the

[11] See [25] §6.2 point 4, and `TSS_Get_PartSign` from `libtss/tssProto.c:1210` and `libtss/tssLib.c:53`

[12] See [25] §6.2 point 5, and `TSS_GMC_Reply` from libtss/tssProto.c:1378

recipient, this change would not be particularly hard to implement. Nonetheless given our time constraints we chose to overlook this issue as, again, it would have virtually no affect our results.

3.2.12 Behavior

With the fairly lengthy list of changes described above complete, we were ready to implement one of the most important pieces of the model. A behavior process was added to each servant that performed the expected honest behavior required of each node participating in the peer group by the policy matrix. Each node transmits the required amount of data to every other node while its certificate is valid. Only capabilities that may be made use of are exercised by each honest node. Once the necessary data are exchanged, the node requests re-certification from the nodes with which it has interacted during this period.

Each node may be assigned a virtual bandwidth that limits the amount of data it may exchange per period. Based on this bandwidth, and the threshold t, each node calculates the number of hosts with which it *must* interact, and the number with which it is *capable* of interacting while still observing requirements of the policy matrix for each host. Each servant will send to as many as possible to as to tolerate the greatest degree of malicious behavior during re-certification, but will not spread itself so thin as to jeopardize meeting its obligation with each other servant.

Each servant tries to wait until it has met its obligation with all of the hosts with which it is interacting. Should its certificate become in danger of expiring, the servant will cease waiting and request its GMC from the hosts with which it has already met the obligation imposed by the policy matrix. If its certificate should be in danger of expiration, and it has *not* met this obligation for some reason, the servant continues on in hopes that it will satisfy the matrix in time. Should it prove unable to do so, it simply gives up.

GMC requests are made only to hosts with whom the servant has met its obligation; to do otherwise is considered misbehavior. Any misbehavior by a solicited host during the re-certification process is tolerated by way of a restart of the process with the host removed and blacklisted by the servant.

When a sufficient amount of time has passed, according to the policy matrix, a node may request an upgrade to the next level. This process is the same as re-certification, but where the extension *Level* is simply incremented. Before requesting an upgrade to the next level, a servant will check to ensure that it can meet the requirements of that level by analyzing the requirements of the policy matrix at this level. If, given bandwidth constraints, it is unable to operate in the network in the way demanded by the policy matrix, the servant will disclose that it is limited to its present level, and continue to operate at the same level ad infinitum.

In addition to the honest behavior described above, we implemented a number of misbehaviors based on those modeled in § 2.5. Each behavior is enumerated, and by providing the necessary commandline, a host may be instructed to act out a particular misbehavior rather than act honestly. Further details are discussed in § 4.2.

3.2.13 Summary

Our work was aimed at providing a robust implementation of the model we describe in § 2. At times, implementation of the model was found to be held back by limitations of the underlying toolkit, many of which were corrected during the implementation process and are described in the preceding sections. Some of these limitations remain in the implementation as it stands at the conclusion of our work, however as implemented we were certainly able to generate the results we are interested in, and validate the model based on our work. We proceed to discuss this validation process, and the results and the experimental testing process used to generate them in the following chapter.

Chapter 4

Experimental Testing

With the implementation of our Peer to Peer file sharing network complete, we constructed a series of laboratory tests to assess the performance and correctness of our implementation. The tests took two forms. First, in order to assess the performance of our implementation, timer routines were integrated into the code base at different points. These routines were used to determine the resource utilization, the feasibility of the security model implemented, and the actual performance of the implementation. Figure 4.1 illustrates the set of measures considered.

- Time between GMC Request and GMC Acquisition at each level.
- Time taken by each peer node to node to partially sign each GMC Request.
- Time taken between request of a share and receipt of the share.

Figure 4.1: List of Operations Measured

Second, a series of validation tests were designed to validate the correctness of the model. They were designed in such a way that testing will validate the behavior of an honest node when faced with dishonest behavior. These tests had already been written into the codebase as an essential part of the implementation of the model, largely as part of our Monitor module (see § 3.2.8). Testing them required implementing various forms

of misbehavior, which we did. In § 4.2 we discussed the details of this portion of the experiment.

4.1 Experimental Setup

Our performance tests were conducted in a laboratory setting as described below. The validation test were performed locally for a number of group sizes and are described in § 4.2. The reason for this difference was a time constraint on the use of the lab. It is worth noting, however, that as the validation tests simply confirmed correct behavior, it is highly unlikely that the behavior differed in any way because the tests were done locally. The same certainly can not be said for the performance tests, which were necessarily done in the lab.

30 lab machines were available for our tests, each equipped with an Intel Pentium © 3 processor, 128 megs of RAM, running Linux kernel version 2.2.x. We settled on testing groups of size 5, 10, 15, 20, 25, and 30 nodes given the available lab environment. A modulo of 1024 bits was used with the TS-RSA algorithm. A series of scripts were written that allowed for the automated creation of our test cases, and which automatically installed the correct test case on each system. The CA and GA were each run on a system on which a node was also present. Being as lightweight as they were it was thought that their impact would not affect our results.

For the purposes of our experiment, we assumed that 100 bytes was the average outgoing bandwidth of the nodes being tested. This assignment allowed us to construct Group Charters with the properties discussed in § 2 and limit behaviors like Masquerading. We assigned this virtual bandwidth, which we will call B, to each node. Recall the Group Charter discussed for our P2P network in § 2.4. As discussed in § 3, each node tried to send to as many Trusted Domain members as it could within the bounds of B.

Group Charters for each of the test groups were constructed by the following criteria. For the Bytes element of our policy matrix, we required each host to commit 35% of B at Level 0, 45% at Level 1, 55% at Level 2, and 65% at Level 3. The rationale for this selection was that it would burden each node sufficiently to dissuade attempts at masquerading while not overburdening them to much as to be completely impractical. Recall that the *Bytes* field of the Group Charter for our P2P network is the total number of Bytes that a node a must have received from a node b for a to sign a GMC renewal request from a. The calculation of this field was done as follows, using level zero for our example.

First, we took $(B/T) \cdot p\% = X$, as the number of bytes that a node must have contributed to any host in the group for its GMC request to be approved, and where p is the percentage used for each level. Then, taking R to be the number of seconds before a must renew its certificate, we computed $Y = X \cdot R$ as the total number of bytes that a must send to each host for each renewal period. Upon receipt of a GMC request, the host b can simply check whether a has uploaded at least Y bytes, and if so, the request proceeds.

The *Renewal* period was chosen as 30 seconds for every level in the experiment in order to allow nodes to quickly reach the highest level and produce the measurements we were looking for. Membership in the next level was computed as $3 \cdot R$ for all levels above the first level, and four rounds for the first level, again letting R be the renewal period. So, each upgrade to the next level required three rounds of participation at the previous level, with the noted exception above.

The constructed Group Charter for the group of 15 node is presented in Table 4.1 as an example.

Times T_0 to T_1 were selected as appropriate given the size of the group to be tested and the amount of time needed to start the servants. Once the Trusted Domain was established,

Level	Bytes	Searches	Renewal	Membership
0	540	0	30	0
1	1380	3	60	120
2	2520	5	90	300
3	3960	10	120	570

Table 4.1: Group Charter for 15 Node Group

and T_1 elapsed, additional nodes were brought into the group every few seconds in order to distribute the acquisition times to some extent. Over time, varying delays in acquisition further distributed the expiration times, and had the effect of evening out GMC requests across each renewal period. A number of Trusted Domain nodes was selected that was thought to appropriately represent the proportion of TD nodes that might be present for a particular group size. This number would vary for each Group based on their selection of a Group Charter.

Our source package contains a number of scripts used during testing which can be found in the `inst` directory. Each of these scripts generates one of the test groups described above by setting the appropriate parameters for the group, and calling the `setup.sh` script. The `AUTO` variable is set in each script, causing the setup script not to prompt for any unknown values and set reasonable defaults when necessary. The output resulting from execution of one of the test scripts is a test directory that can be copied on to, or share between, the nodes that will constitute the test group. Once run, the test group will generate timing measurements which will be logged to various files. See Appendix A for a complete explanation of this process.

Each of our test cases was run until all nodes had attained the highest level in the Group Charter, which was level 3 for all groups. The Trusted Domain was then brought down, and the data collected and archived. The data acquired during testing was significant in both size and scope. A presentation and analysis of the results can be found in § 5.

4.2 Validation Tests

Our validation test component was quite different, but no less important, than the performance testing. Through our validation tests we sought to confirm the proper functioning of the implementation by observing the reaction of honest nodes to dishonest behavior; specifically to those misbehaviors we discussed in § 2.5. We implemented the three categories of behavior discussed earlier: Malicious, Greedy, and Masquerading. Any node in the network could be directed to exhibit any behavior at runtime; by default nodes acted honestly.

Each of the three types of behavior above was enumerated; the list is provided in Table 4.2 for reference. For each of these three behavior types, the particular ways in which a node exhibiting that behavior might misbehave was also assigned a number. For example, a Greedy node might try to send too many search requests, or upload too little, both of these were enumerated uniquely.

Behavior Type	Number
Honest	0
Malicious	1
Greedy	2
Masquerading	3

Table 4.2: Number Assigned to Each Behavior Type

By providing the behavior type and misbehavior number on the command line[1], any node could be directed to misbehave in a specific way and at a specific time. Tables 4.3, 4.4, and 4.5 list the numbers that were chosen for each behavior. A battery of tests was run during which each of the behaviors was assigned to an appropriate node, and the reaction of honest nodes during misbehavior was observed. In each table, the *Expected Result* is the expected reaction of an honest node to the listed behavior. In all cases, the actual result was the expectation, confirming the validity of this part of the implementation.

[1] See Appendix A for a thorough explanation of how to exhibit this behavior

Each of the misbehaviors listed above required a unique implementation, however some misbehaviors are always tested for by our implementation. Any P2P Protocol packet must contain an accompanying GMC which is validated upon receipt of the packet. If the GMC has expired, is not yet valid, or is not present, the packet is rejected, the behavior is noted, and the sender is blacklisted. Packets are also checked for proper formatting. If an improperly formatted packet is received, it is rejected just as above, and the behavior is noted. These behaviors had been observed and tested for throughout the development and testing process; it was not necessary to write an implementation for these as was required by the other more specific misbehaviors. Table 4.3 summarizes the above.

When	Behavior	Expected Results
Always	Invalid or Missing GMC Improperly Formatted Packet	Packet Rejected, Sender Blacklisted

Table 4.3: Implicit Validation Tests

4.2.1 Malicious Validation Tests

We decided that the majority of misbehavior qualified as Malicious. Table 4.4 summarizes the behavior that a malicious node was allowed to exhibit. The behavior number is listed in the first column; the column *When* determined when each behavior was exhibited. Each of these behaviors was assigned to a node at the appropriate time, and the reaction of honest nodes to each behavior was observed to be as expected (see *Expected Result*).

Test Number	When	Behavior	Expected Result
1	T_0 to T_1 (Bootstrap)	Request early Start	GMC Request Denied, Blacklisted
2		Request excessive Expiration	
3		Request inappropriate Level	
4	Any Time After T_1	Attempt to Bootstrap	GMC Request Denied, Blacklisted
5		Request early Start	
6		Request excessive Expiration	
7		Request inappropriate Level	
8	Prior to any GMC Request	Attempt to Search at Level 0	GMC Request Denied, Blacklisted
9		Attempt to Download at Level 0	
10		Attempt to Download at Level 1	
11		Attempt to Download at Level 3	

Table 4.4: Validation Tests for Malicious Behavior

4.2.2 Greedy Validation Tests

The behaviors listed in Table 4.5 were decided to be Greedy. Just as with the Malicious behaviors, each of those behaviors listed as Greedy were assigned to a node at runtime, and the reaction of honest nodes to each behavior was observed to be as expected (see *Expected Result*). Again, see § 3.2.12 on the details of how behaviors were implemented and Appendix A for details on how they can be assigned to a node.

Test Number	When	Behavior	Expected Result
1	Prior to any GMC Request	Excessive Search requests	GMC Request Denied, Blacklisted
2	Before any GMC Request	Failure to meet Bytes requirement at Level 0	GMC Request Denied, Blacklisted
3		Failure to meet Bytes requirement at Level 1	
4		Failure to meet Bytes requirement at Level 2	
5		Failure to meet Bytes requirement at Level 3	

Table 4.5: Validation Tests for Greedy Behavior

4.2.3 Validation Test for Masquerading

Testing the behavior of a node attempting to Masquerade required a slightly different approach, but was also a part of our validation testing. Recall from § 4.1 and § 4 that each node could be assigned a virtual bandwidth that it would not exceed. Recall as well from our discussions of a P2P file sharing network using the model in § 2.4 the way in which the Group Charter was constructed. Given the choices made in the policy matrix, no single host should be able to transmit two times the amount of data required at the highest level.

In order to test that the implementation properly limited Masquerading as the model intended, we selected two nodes a and b out of each test group and claimed that they belonged to a single host which wished to masquerade. The total bandwidth of this host would be insufficient to fulfill the requirement of the highest level twice. Nontheless, it was distributed between nodes a and b. Node a was given a sufficient fraction of the bandwidth to allow it to reach the maximal level in the group, while b was left with the remainder.

The Trusted Domain was then formed, just as during performance testing, and a and b were run and directed to masquerade as described above; their behavior otherwise was that of an honest node. Recall from § 3.2.12 that prior to requesting an upgrade to the next level, each node tested to see that it could fulfill the requirements of the Policy and Capabilities matrices at that level. Since node b was given insufficient bandwidth to reach the highest level, eventually it was unable to pass this test. Our implementation for this behavior largely consisted of an alert to the user that the node b had become permanently stuck at its present level. It was able to continue participating in the network, but as expected, masquerading was limited.

Our implementation behaved as expected in the presence of Malicious, Greedy, and other behaviors. Our observations during this testing period all coincided with the *Ex-*

pected Result portion of Tables 4.3, 4.4, and 4.5 from the previous sections. As directed earlier, the reader can refer to Appendix A for a thorough description of how the tests can be run. In the next section, a detailed analysis of our results from the first portion of our experiment can be found.

Chapter 5

Results and Analysis

In the previous section we detailed our performance testing methodology, and described how we performed measurements of various key parts of our implementation. We now present the details of those results, as well as a detailed analysis intended to reveal important trends and facts about our work. We begin by presenting a summary of the results.

5.1 Summary of Results

Group Size	5	10	15	20	25	30
Threshold	2	4	6	7	8	9
GMC Acquisition	0.501756	1.136846	1.698270	2.056688	2.447770	2.893956
Partial Signature	0.157661	0.165857	0.168316	0.174676	0.181019	0.187949
Share Acquisition	0.031923	0.068558	0.136361	0.221222	0.328974	0.364317
Partial Share	0.007545	0.009190	0.010694	0.011657	0.012678	0.013203

Table 5.1: Median of All Performance Measurements for all Thresholds and Group Sizes

Table 5.1 summarizes our laboratory measurements for all thresholds and group sizes, where all values are expressed as the median of the observed results. In this table, we present the following. *GMC Acquisition* is the time taken to acquire GMC_{new} for some threshold t. *Partial Signature* is the time taken to issue a partial signature on GMC_{new} for some threshold t. *Share Acquisition* is the time taken to acquire a share by some

highest-level node for some threshold t. Last, *Partial Share* is the time taken by some highest-level node to create a partial share.

By far the greatest time taken above is for the GMC acquisition process, which involves the greatest number of message exchanges, and is perhaps the greatest computational burden of all the recorded processes. Not unsurprisingly, the operation that took the second greatest amount of time, from Table 5.1, was Share Acquisition. This process, similar to GMC acquisition, involves a number of message exchanges, being an aggregate of a number of Partial Share computations and message exchanges. Both Partial Signature and Partial Share times, by comparison, were quite small. We explore these and other observations about our result in the following sections.

5.1.1 GMC Acquisition Results

GMC Acquisition time is an important benchmark for the performance of our system. Besides being a common operation, and therefore one which should require as little time as possible to complete, acquisition time is significant because in large groups, given a sufficiently short renewal period, if this process takes too long a node will be unable to acquire GMC_{new} before its certificate expires.

Group Size	5	10	15	20	25	30
Threshold	2	4	6	7	8	9
GMC Acquisition	0.501756	1.136846	1.698270	2.056688	2.447770	2.893956

Table 5.2: Median GMC Acquisition Time for all Thresholds and Group Sizes

Table 5.2 shows the median time for a node to successfully complete the entire GMC acquisition process listed by Threshold. Measurement for this metric began before the node submitted its first Sign Request (see § 3.2.9), and ended when the GMC had been successfully acquired and verified. This table appears as a graph in Figure 5.1. For each group, the corresponding threshold selected can be found in Table 5.2.

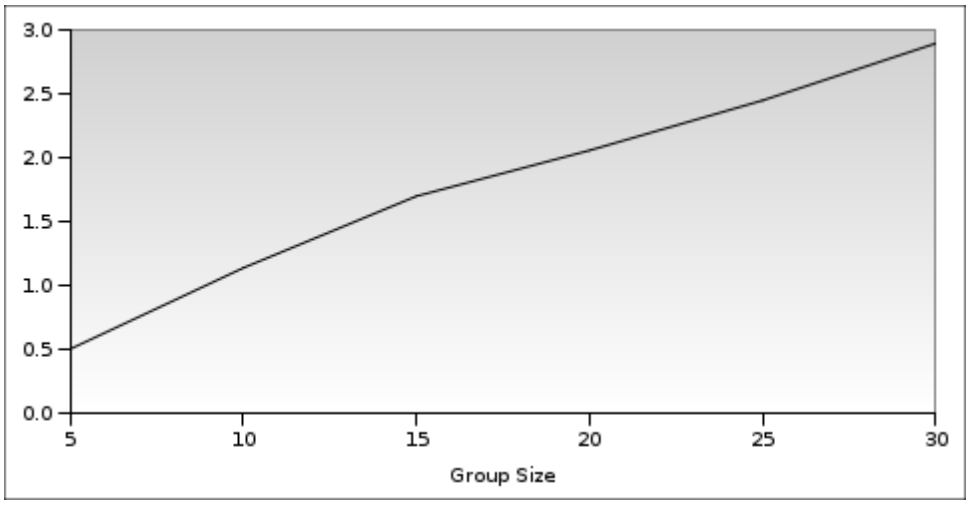

Figure 5.1: GMC Acquisition Time (sec) vs. Group Size

Figure 5.1 shows what appears to be essentially linear growth of the median time taken to acquire a GMC at any level for all group sizes. In Figure 5.2 we break this result

down by level, plotting the median GMC acquisition time at each level for each of the groups measured.

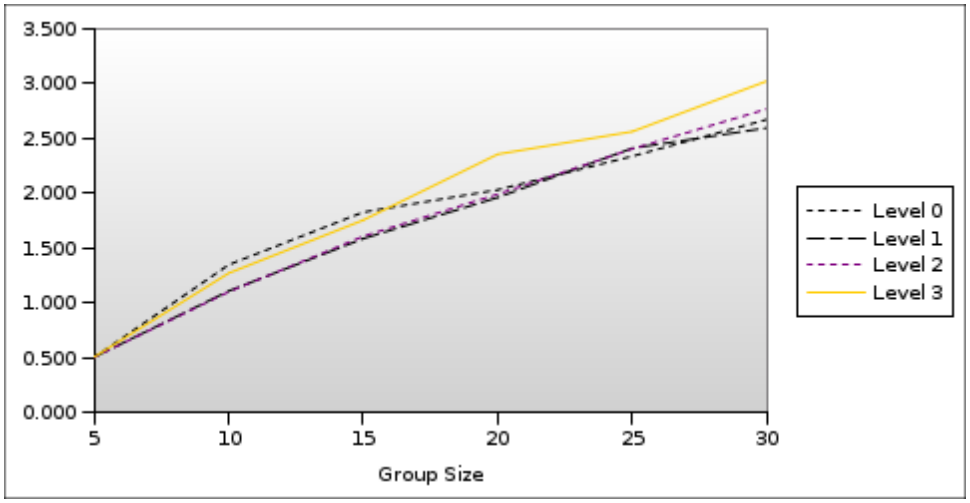

Figure 5.2: GMC Acquisition Time (sec) by Level vs. Group Size

Note that acquisition at level 3 takes noticeably longer than at the previous level. Acquisition at level 2 can also be seen, with somewhat more difficulty, to be greater than that of level 1. In fact, for each increase in level, there is a slight increase in the median GMC acquisition time. We explore this result later in this section.

5.1.2 Partial Signature Results

As we saw previously, for each successful GMC request, at least t Trusted Domain members must issue a partial signature on the requested GMC_{new}. Here we examine the results of our measurement of the partial signature time.

Group Size	5	10	15	20	25	30
Threshold	2	4	6	7	8	9
Median Sign Time	0.157661	0.165857	0.168316	0.174676	0.181019	0.187949

Table 5.3: Median GMC Signature Time for all Thresholds

The median Partial Signature time for all thresholds appeared earlier in Table 5.1 and is repeated here by itself in Table 5.3 for convenience. Measurement of this result began upon receipt of a valid signature request, and ended once GMC_{new} had been successfully partially signed. A graph of these values appears in Figure 5.3.

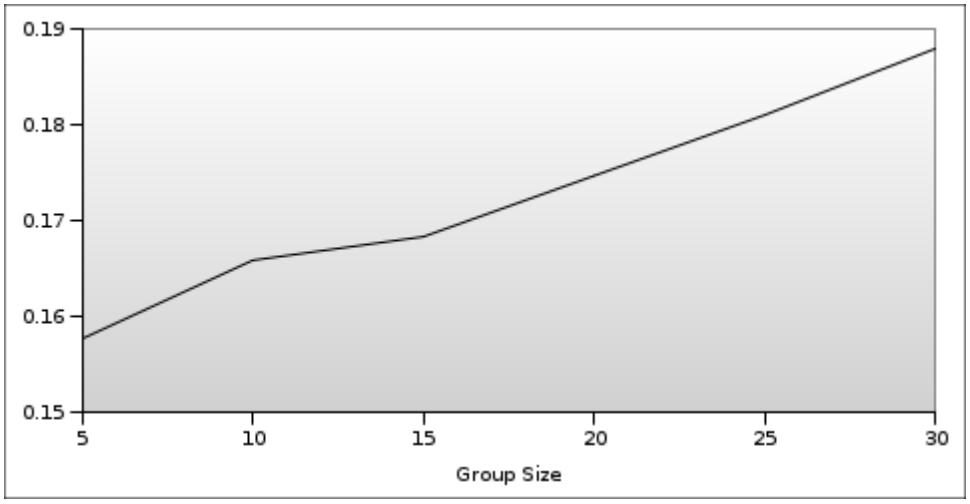

Figure 5.3: Median Time to Sign GMC (sec) vs. Group Size

The growth of the median signature time as group size and threshold increases appears essentially linear, and the time taken appears quite small; consider that as t is tripled between the group of 5 nodes and that of 30, the median signature time increases by less than 0.03 seconds.

5.1.3 Share Acquisition Results

The performance of partial share acquisition and share creation time were both measured. Measurement of this metric began when a host started to solicit the necessary t Trusted Domain members for partial shares, and ended when the shares had been reconstituted into a full share of the group secret. In Table 5.4 we see our observations of share acquisition time.

Group Size	5	10	15	20	25	30
Threshold	2	4	6	7	8	9
Share Acquisition	0.031923	0.068558	0.136361	0.221222	0.328974	0.364317

Table 5.4: Median of All Performance Measurements for all Thresholds and Group Sizes

In Figure 5.4 we plot the Share Acquisition time by group size for all levels. Note that this is an essentially linear operation that grows slightly for each increase in the Threshold, but remains below even half a second for the greatest threshold recorded. Consult Table 5.4 for the threshold used for each group.

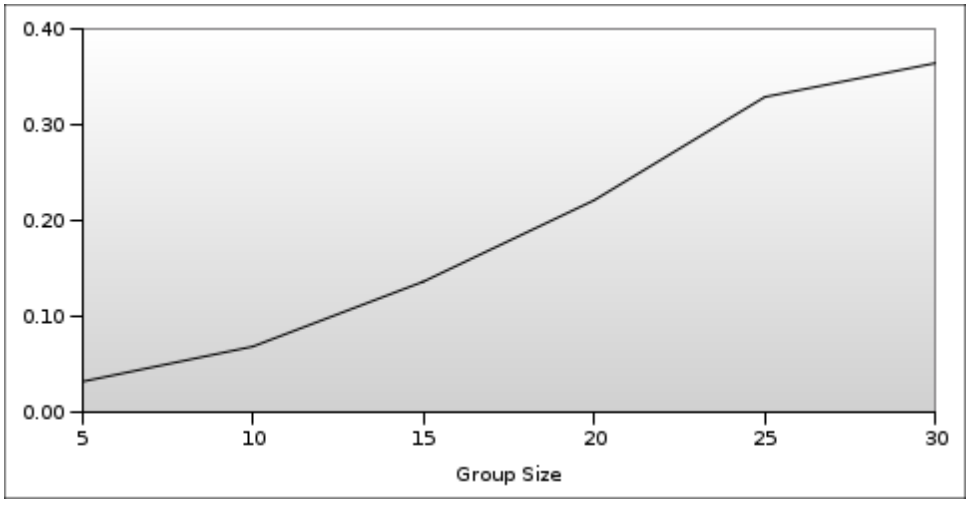

Figure 5.4: Median Share Acquisition Time (sec) vs. Group Size

Share Acquisition times are not nearly as crucial a measurement as compared to GMC acquisition time in our work. In our implementation, a share is only acquired once per

host for the entirety of its participation in the group. We now examine the time taken for *issuance* of partial shares.

5.1.4 Partial Share Issuance Results

The results from the previous section detailing the total time taken to acquire the necessary t partial shares. This time was an aggregate of the operation that we now concern ourselves with, the time taken to issue a partial share for each of the t respondents, above. Measurement of this quantity began once a partial share request had been received, and ended once a share request was fulfilled, and a new partial share issued. In Table 5.5 we see our results from observation of the partial share issuance process.

Group Size	5	10	15	20	25	30
Threshold	2	4	6	7	8	9
Partial Share	0.007545	0.009190	0.010694	0.011657	0.012678	0.013203

Table 5.5: Median Share Issuance Time for all Thresholds and Group Sizes

In Figure 5.5 we plot the time taken to issue each Partial Share by group size for all levels. Note that, as with Share Acquisition time in the previous section, this is an essentially linear operation that grows slightly for each increase in the Threshold, but takes a total time that is marginal; for the greatest threshold recorded, this operation took only slightly over a one-hundredth of a second. The threshold used for each of the groups in Figure 5.5 can be found in Table 5.5.

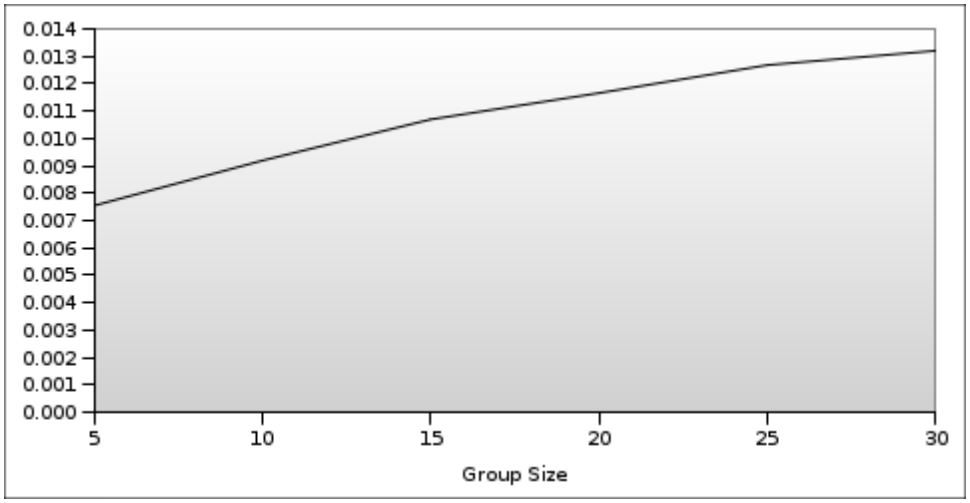

Figure 5.5: Median Share Issuance Time (sec) vs. Group Size

Both partial share and partial share issuance occurred very quickly in our implementation, especially by comparison to the GMC times. Furthermore, while GMC acquisition is the most frequently occurring operation measured, partial share acquisition is the least frequently occurring, as it happens at most once for every host in the peer group that manages to become a member at the highest level.

5.2 Timing Analysis

In the previous sections we concluded that the most significant operation of those measured was the GMC Acquisition process, given both the time it took to complete, and the frequency of the operation compared to others. In this process, t partial signatures on some GMC_{new} were issued. This operation was clearly the most expensive of those observed, and given its importance to the overall functioning of the model and implementation, we wish to explore it more completely.

In this section we endeavor to reconstruct the process of GMC Acquisition as the sum of its parts. We construct a formula expressing the timing of the process, and use it to approximate our own results, in order to determine its accuracy. Later, we make use of this formula again in approximating results for group sizes and thresholds greater than those that were observed during experimentation.

5.2.1 Overview of GMC Acquisition

Figure 5.2, which graphed the GMC acquisition time for all group sizes by level, showed that while apparently linear, the GMC acquisition time in fact grew slightly for each increase in the threshold, t. Let us explore this result.

Recall that for each successful GMC acquisition, t partial signatures must be obtained. This process occurs as follows in our implementation[1]. First, a node A solicits at least t members of the Trusted Domain, asking them to commit to signing its GMC. Then, A constructs a *Commit List* containing t nodes out those nodes who agreed to sign. As stated by Narasimha et al. in [25] this preliminary round is necessary to establish the list used in Lagrange Polynomial coefficient calculation[2].

Following construction of its *Commit List*, A contacts each host on the list, requesting

[1] It occurs in the same manner in the Bouncer toolkit.

[2] The construction of this list is addressed in § 5.3 where improvements to the process are suggested

that they certify its GMC_{new} by issuing the required partial signatures. Once t partial signatures are received A sends its request for a GMC to the GA, which responds with the new GMC for A[3].

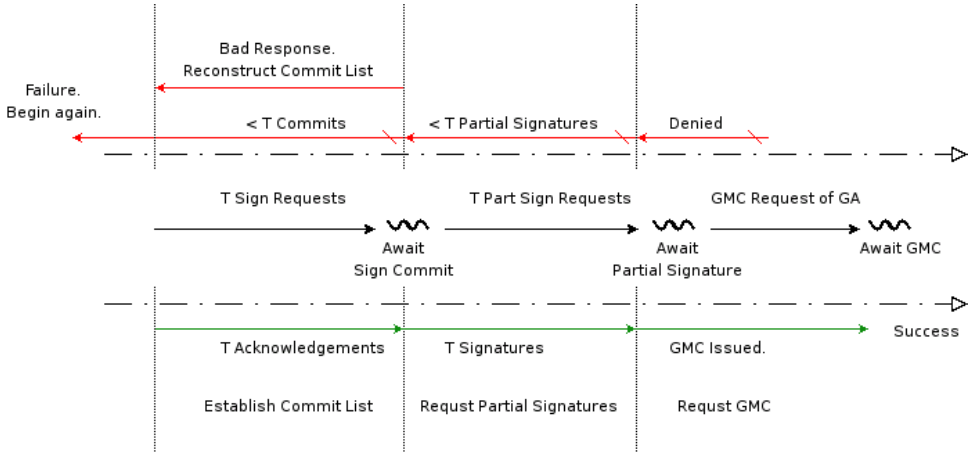

Figure 5.6: A Timing Diagram for the Process of GMC Acquisition

A timing diagram for the acquisition process can be seen in Figure 5.6. Encapsulated between two long horizontal arrows from left to right lie the communication rounds that A must go through in acquiring GMC_{new}. Below this is a description of the rounds that must be completed successfully for each GMC acquisition. Failure during the process can occur for any of the reasons seen above at the top in red, and forces A to restart the process of acquisition.

So, given this description of the process, and the diagram seen in Figure 5.6, we can say that the GMC acquisition process is the sum of several operations. Most significantly, each successful acquisition is comprised of the t partial signatures that are issued. Second, and also important, is network and packet processing delay that occurs for each of the message exchanges. Finally, two exchanges contribute to the process which are roughly constant time: the *Commit List* exchange, and the final exchange in which the GMC is acquired from the GA.

[3]We explore this dependency in § 3.2.11.

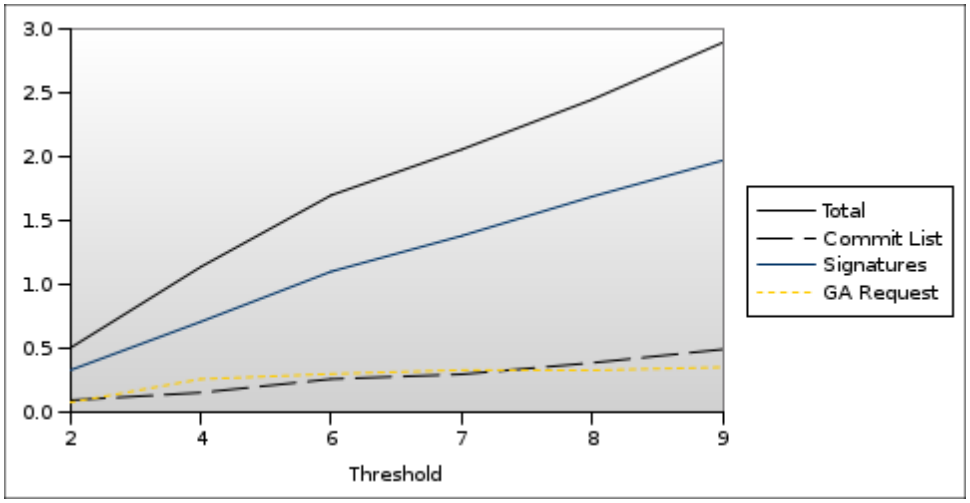

Figure 5.7: Time Taken by Exchanges in GMC Acquisition (sec) vs. Threshold

In Figure 5.7 we see the time taken (in seconds) by each of the exchanges in the GMC acquisition process plotted against the total time taken. We can draw a number of conclusions from this graph. First of all, our earlier claim that the two exchanges we discussed above as being roughly constant time proportional to t is clearly legitimatized. Secondly, the greatest contributor to GMC acquisition time is in fact the partial signature exchanges as we discussed above. Clearly neither of the constant-time message exchanges above is a significant contributor to the acquisition process when compared to signature time.

Bearing this conclusion in mind, we proceed to the next section in which we break the GMC message exchanges down precisely

5.2.2 Overview of Timing Analysis

We begin our construction of a formula expressing the GMC acquisition process by looking at the greatest contributor to this process, the issuance of Partial Signatures. As we have said, t such partial signatures are issued for each successful acquisition. Let us call the time taken to issue a single partial signature t_{sign}, and the aggregate time taken to issue the necessary t partial signatures for each acquisition $sign_{total}$. Finally, for the purposes of our exploration, let us factor out the GA and *Commit List* exchanges, as well as network and packet processing delays, as the constant t_{const}. Bearing these terms in mind, we can express the total time taken by some GMC acquisition, $t_{acquire}$, as in Equation (5.1).

$$t_{acquire} = (sign_{total} \cdot t) + (t_{const} \cdot t) \tag{5.1}$$

5.2.3 Timing Analysis of Partial Signature

Importantly, it turns out that we can describe $sign_{total}$ from Equation (5.1) with greater precision. First, recall that each signature is generated by some node v_j present in the *Commit List*, $v_j \in (v_1, v_2, \ldots, v_n)$. From Equation (1.5) in § 1.1, we know that each Partial Signature on some GMC_{new} submitted by some node v_j is calculated as $m^{d_j} \mod N$. Here, the message m is GMC_{new}, the partial secret key d_j is computed as $ss_j \cdot l_j(0)$, and ss_j is the secret share being used[4].

The key to expressing $sign_{total}$ with greater precision lies in an observation that we can make regarding the computation of $l_j(0)$, which is as follows[5] [25].

$$l_j(z) = \prod_{i=1, i \neq j}^{t} \frac{z-i}{j-i} \pmod N \tag{5.2}$$

[4]See § 3.2.11 regarding an important disparity in partial signature generation on GMC_{new} as implemented in the Bouncer toolkit version `0.7`

[5]Here, j is the index v_j from the *Commit List*.

For each increase in t, the product l_j increases by $\frac{z-i}{j-i} \pmod N$. Observe that in Equation (5.2) the product $l_j\,(z)$ is made up of $t-1$ rounds of computation. Let us do the following. First, we factor out the increase in l_j per level as the term δ_{sign}. Bearing in mind that this term is an increase proportional to t given some basis time, let us say that in the absence of this increase, we have the basis term t_{sign}. Putting the two together, we can construct Equation (5.3).

$$sign_{total} = t_{sign} + ((t-1) \cdot \delta_{sign}) \tag{5.3}$$

Given the observations we have made during experimental testing of our implementation, we can calculate and assign values to the terms in Equation (5.3) and ultimately Equation (5.1) based on our results. In the next section, we begin to do so by assigning values to the terms in our equation for $sign_{total}$.

5.2.4 Determining t_{sign} and δ_{sign}

To begin to determine values for t_{sign} and δ_{sign}, let us examine the total time taken for both signing and acquiring a GMC across all group sizes. These values can be seen in Table 5.6.

Threshold	Signature Time	Acquisition Time
2	0.157661	0.501756
4	0.165857	1.136846
6	0.168316	1.698270
7	0.174676	2.056688
8	0.181019	2.447770
9	0.187949	2.893956

Table 5.6: Signature and Acquisition Time for all Thresholds

Now we look at the difference in time required to sign a GMC between different group sizes and different thresholds. Table 5.7 shows the difference between signature times for the increasing thresholds used.

Difference Between Thresholds	Signature Diff
2 and 4	0.004098
4 and 6	0.001230
6 and 7	0.006359
7 and 8	0.006344
8 and 9	0.006930

Table 5.7: Difference in Signature Time Between Groups With Adjacent t

Each value in the *Signature Diff* column is the increase in time taken to sign from one threshold to the next. If we select the median from Table 5.7, we then have the median increase in signature time for each increase in t. Recall that this is what we said δ_{sign} represented, the increase in signature time proportional to t. So, selecting the median as the value for this term we can say that for our data, $\delta_{sign} = 0.006344$ seconds.

We know that δ_{sign} appears in the term $sign_{total}$ exactly $t - 1$ times. So, δ_{sign} should be present in each of the signature times from Table 5.6 $t - 1$ times, once for each round

of computation during Lagrange Polynomial coefficient calculation. We can remove the term δ_{sign} from $sign_{total}$ by subtracting $(t-1) \cdot \delta_{sign}$ out of $sign_{total}$.

Table 5.8 shows the original signature times (again for groups with adjacent values of t) next to which we see this time with the term δ_{sign} removed as suggested above.

Threshold	Signature Time	Signature Time Without δ_{sign} Term
2	0.157661	0.151318
4	0.165857	0.146826
6	0.168316	0.136599
7	0.174676	0.136615
8	0.181019	0.136615
9	0.187949	0.137201

Table 5.8: Breaking δ_{sign} Out of Signature Time

Taking the median of the column *Signature Time* from Table 5.6 yields the value 0.136615. We can use this value as the basis for any sign operation, which we have said is represented by the term t_{sign}.

In summary, through some simple arithmetic we have calculated the two values of interest based on our sample data as we see them in Figure 5.8.

Figure 5.8: Values of δ_{sign} and $sign_{total}$

δ_{sign}	$= 0.006344$
$sign_{total}$	$= 0.136615$

5.2.5 Extrapolation Using $sign_{total}$

Given our assessment of the terms from Equation (5.3), we proceed to make use of these values in approximating two things. First, we approximate our own data using this equation, and determine the percent error resultant from a comparison to our actual results. Second, given that this comparison legitimizes our results for $sign_{total}$, we proceed to approximate signature times for group sizes larger than we were able to experimentally measure.

Table 5.9 calculates the approximation using $sign_{total}$, and shows it against the actual value along with the percent error between the two.

Threshold	$sign_{total}$	Actual Sign Time	Percent Error
2	0.142958	0.157661	9.33%
4	0.155645	0.165857	6.16%
6	0.168332	0.168316	0.01%
7	0.174676	0.174676	0%
8	0.181019	0.181019	0%
9	0.187363	0.187949	0.31%

Table 5.9: Approximated Sign Time and Actual Sign Time

A plot of these values can be seen in Figure 5.9. The greatest percent error can be seen the groups with t values 2 and 4 respectively. The reasoning for this is that first, as the group size (and t) increases, the number of samples increases, and so the median Signature Time becomes more accurate. Second, as t increases, the number of rounds during Lagrange Polynomial coefficient calculation increases, yielding a more accurate value for δ_{sign}.

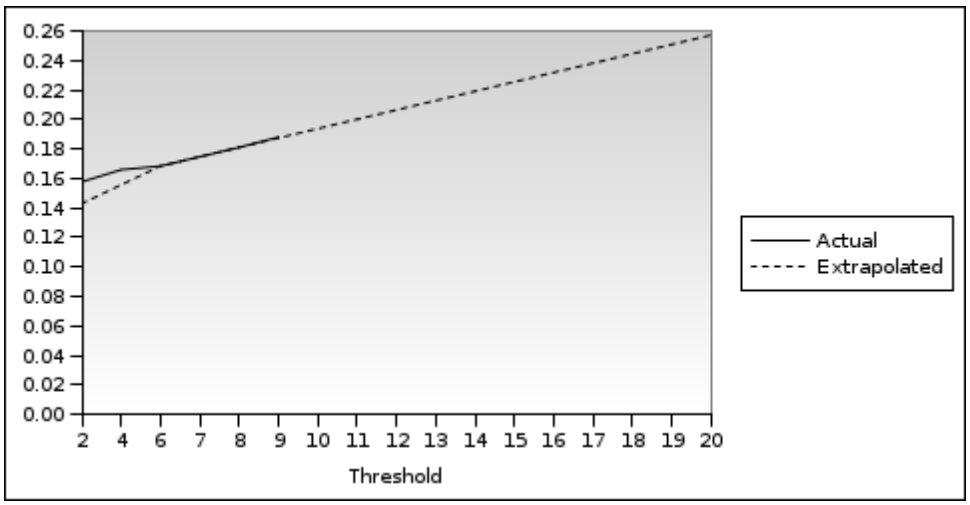

Figure 5.9: Predicted GMC Sign Times (sec) vs. Actual (sec)

Based on the closeness of the fit seen between actual and approximated Partial Signature times in Figure 5.9, as well as the low percentage of error observed in Table 5.9, we can conclude that our result for $sign_{total}$ is an accurate estimation of Partial Signature time.

Looking at the graph in Figure 5.9, we can see that even for thresholds of over twice those measured during our experiment, signature times remain as low as a quarter of a second. Between the least and the greatest of the thresholds used, above, we see that the total increase in $sign_{total}$ was less than a tenth of a second. Note that, by contrast, we have already concluded that the most significant contribution to the acquisition process was $sign_{total}$.

Two factors account for this discrepancy. Recall that t instances of the term $sign_{total}$ contribute to $t_{acquire}$, meaning that the values seen in Table 5.9 appear artificially low. Multiplying each by the threshold t gives a better idea of how much time the $sign_{total}$ operation consumes. Furthermore, each of these instances brings with it transmission and packet processing delays at both the send and receive end of the exchange. As we will see later, in computing t_{const}, these delays become significant as t increases.

5.2.6 Calculation of t_{const}

As we mentioned earlier, the constant t_{const} from Equation 5.1 represents several constant operations proportional to t for some group size. First, and most importantly, network and packet processing delays are represented here. Second, the roughly constant-time operations for GA and *Commit List* exchanges. We now endeavor to assign a value to this term based on our data.

We can easily calculate the actual values for t_{const}; that is, the actual time taken by the operations that we have said contribute to t_{const}. Recall that for each successful GMC acquisition, t partial signatures must be acquired (Equation (5.1)) and that therefore, t instances of the term $sign_{total}$ are present in each GMC acquisition time. For each of the median GMC Acquisition times, then, if we subtract out $sign_{total}$ exactly t times, we have arrived at the total time required by the operations in t_{const} for that threshold. Dividing by t we arrive at the value of t_{const} proportional to t.

We call this computation, which appears in Equation (5.4), $actual\_t_{const}$ to differentiate it from t_{const}, although we note that ultimately they represent the same terms.

$$actual\_t_{const} = t_{total} - (t \cdot sign_{total}) \tag{5.4}$$

In Table 5.10 we perform this computation, and display it alongside each of the variables used above. Note that in this table, we have used the observed median values for *Signature Time* for each threshold rather than our approximation using $sign_{total}$ so as not to introduce greater error into the calculation.

We can use Table 5.10, in particular the computed values of $actual\_t_{const}$, as the basis for our approximation, t_{const}. It turns out that a rather simple approach to computing this term yields an accurate result. Because the operations constituting t_{const} are all linear proportional to t, we can represent t_{const} as line of the form $y = m \cdot x + b$, where $x \equiv t$.

Threshold	Signature Time	GMC Acquisition Time	$actual\_t_{const}$
2	0.157661	0.501756	0.093217
4	0.165857	1.136846	0.118355
6	0.168316	1.698270	0.114729
7	0.174676	2.056688	0.119137
8	0.181019	2.447770	0.124952
9	0.187949	2.893956	0.133602

Table 5.10: Determining Values for $actual\_t_{const}$

We simply select $b = 0.093217$, calculate δ_y as the difference between the first and last of our $actual\_t_{const}$ values, δ_x as the difference between the first and last threshold, and finally m as $\frac{\delta_y}{\delta_x}$. Doing so yields Equation (5.5).

$$t_{const} = 0.005769t + 0.093217 \tag{5.5}$$

We present the fit that this approximation yields, against the actual result, by way of presenting our final result for the computation of t_{total} in the following section.

5.3 Extrapolation Using t_{total}

Having computed each of the terms of t_{total} from Equation (5.1), we can finally make use of this result in approximating GMC acquisition times for group sizes and thresholds greater than those measured during our experiment. This is important as it will give us an idea of how the implementation would perform for much larger groups.

Threshold	t_{const}	t_{total}	Actual	Percent Error
2	0.104755	0.495427	0.501756	1.26%
4	0.116294	1.087755	1.136846	4.32%
6	0.127832	1.776986	1.698270	4.64%
7	0.133602	2.157940	2.056688	4.92%
8	0.139371	2.563120	2.447770	4.71%
9	0.145140	2.992524	2.893956	3.41%

Table 5.11: Comparing Extrapolated t_{total} to Actual

In Table 5.11 we compute t_{total} for those thresholds which we have measured during our experiment. *Actual* is the actual time taken to acquire a GMC at each threshold, next to which is a percent error comparing this to the estimate. The greatest percent error can be seen the groups with t values 2 and 4 respectively.

It is thought that two factors contribute to this outcome. First, as the group size (and t) increases, the number of samples increases, and so the median Signature Time becomes more accurate. Second, as t increases, the number of rounds during Lagrange Polynomial coefficient calculation increases, yielding a more accurate value for δ_{sign}. Our approximations of t_{total} can be considered sufficiently accurate given the stable, low percentage of error in the computation as seen here.

Note that the values of t_{const} approach those of $sign_{total}$ which we saw earlier. Consider, for example, that at a threshold of 9, $sign_{total}$ consumed 0.187363 seconds proportional to t, while above, we see that the same result for t_{const} consumed 0.145140 seconds proportionally. And yet, by contrast, in Figure 5.7 we saw that overwhelmingly, $sign_{total}$ was the greatest contributing term to t_{total}.

In trying to understand this disparity, it is crucial to recall that packet and network delays are represented by the t_{const} term. The contribution that the *Commit List* and GA exchanges made was nearly constant. This suggests, then, that the greatest contributing factor to the magnitude of t_{const} was network and packet processing delay.

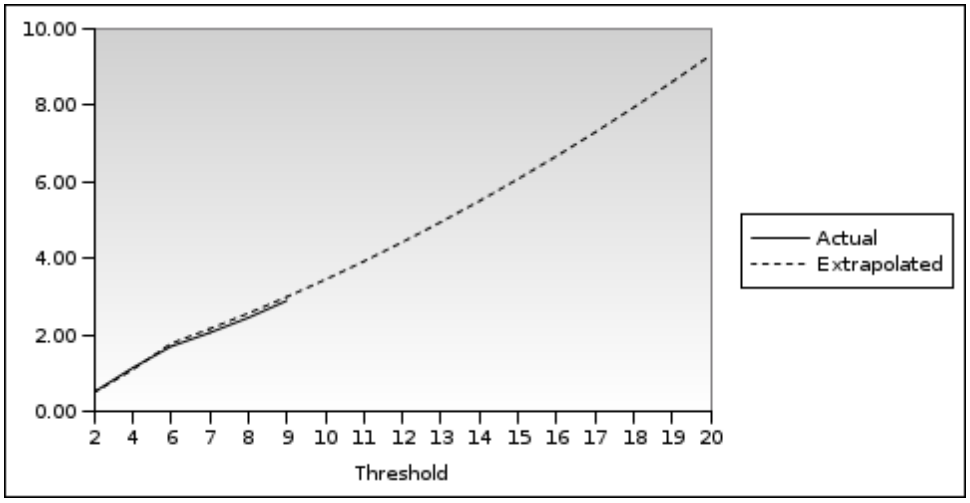

Figure 5.10: Extrapolated GMC Acquisition Time (sec) vs. Actual (sec)

In Figure 5.10 we calculate and plot t_{total} for thresholds equal to, and importantly, greater than those used during our experiment. Note that as the threshold increases to its maximum of 20 in this figure, t_{total}, the total time taken to acquire a GMC, is estimated to be greater than 9 seconds. Recall from § 4.1 that our selection of a renewal period was 30 seconds for nodes participating at level 0. Clearly, therefore, an acquisition time of nearly 10 seconds, almost a third of time a node has to participate before it must re-certify, suggests a number of things.

First of all, while an acquisition of this length may be suitable to some environments, it would certainly not be suited to our implementation as we have discussed it here. It is crucial to point out that in the Bouncer toolkit, as made use of by our implementation, the GMC acquisition process occurs entirely serially. That means that a large number of network and packet processing delays is aggregated into t_{total}. By contrast, if this process were made to be parallel, during signature acquisition, for example, the total delay added

to t_{total} would be the greatest delay in sending, and the greatest delay in receiving to any host on the *Commit List*. Bearing in mind that there are t such exchanges, parallelizing this process would reduce t_{total} by a factor of t, a highly significant savings.

Furthermore, the *Commit List* round of messages can be eliminated reducing the message exchange to a single handshake with t hosts. The justification for this is that each soliciting node is aware of those nodes with whom it has acted in our implementation, and whom it can reasonably trust to partial sign its request. Should a node reject the request, the result is the same as if the *Commit List* round were left in: the node must restart the process. So, in effect, nothing is lost in this reduction. We note that a *Commit List* must still be used in some fashion in the acquisition process, and that it is the message exchange round that may be eliminated here, rather than the list itself.

Lastly, in reflecting on the subject of reducing t_{total} it is useful to point out that, as originally stated in [25], the Lagrange coefficients may be precomputed. We point out that, similarly, and perhaps more importantly in this case, they may also be cached. This could result in significant savings, given the contribution of the $sign_{total}$ term to t_{total}. Caching, in this case, would mean simply remembering the coefficients for some *Commit List* that was used.

5.4 Comparison to Prior Work

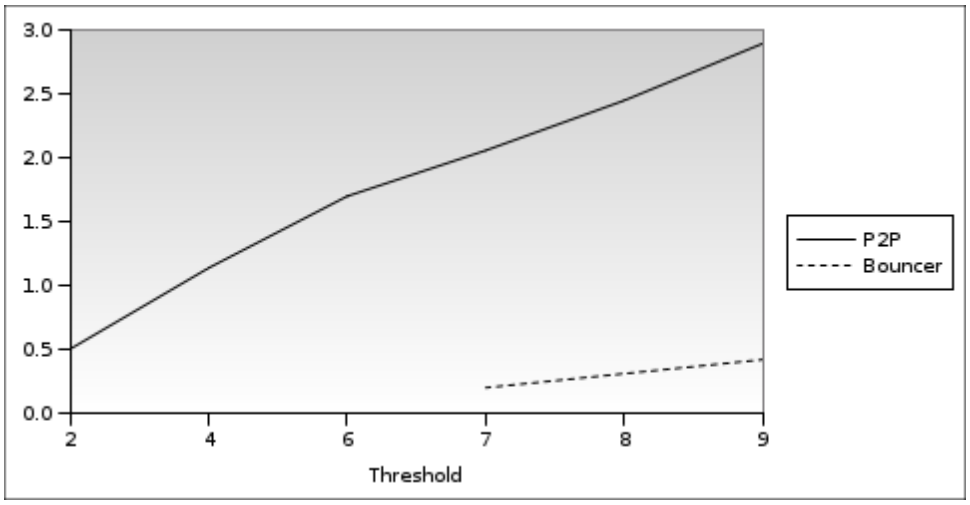

Figure 5.11: GMC Acquisition (sec) vs. Narasimha et al.

In Figure 5.11 we see the median GMC acquisition result for all Thresholds plotted against the same result taken from an approximation of Narasimha et al. [25]. Immediately obvious is that our median acquisition time is much greater than, and grows at a much faster rate than that observed by Narasimha et al.

Several explanations can be offered that account for this result. To begin with, we make the assumption that Bouncer version `0.7`, or its functional equivalent, was used to generate the results seen above, for the purposes of this commentary. Making this assumption is ostensibly reasonable, given that code to generate their results is present in this version of their codebase. Importantly, we note that our implementation is measuring essentially the same thing in this case, which justifies the comparison.

First, by way of explaining the discrepancy between the Narasimha et al. [25] result and that observed during our own testing, we suggest that the increase is due in large part to the load that our clients were under, which caused significant network and packet processing delay. In their implementation, each node is almost certainly idle when any request is received (see § 3.1). In our implementation, at any given time, a node receiving

a GMC request has a substantial packet processing load. In part the discrepancy is thought to be due to the latency between receipt of the packet, and actual processing.

Exacerbating the processing delay is the fact that our implementation maintains a Tracker module and a series of Monitor checks that execute at least once per packet received (see § 3). Each of these extends the turnaround time for any request by performing their own checks, queries, and so forth. Furthermore, each makes use of locking to coordinate access to a shared data repository amongst asynchronous child processes. Locking is necessary in this case, but is thought to add substantial delay overhead.

Lastly, we offer the following remarks. Our implementation was designed for completeness and correctness, and locking mechanisms such as those above tend to require some optimization to minimize their impact; this was not done given time constraints. It is thought that optimization of the existing synchronization techniques would have a noticeable impact on performance in this case.

Chapter 6

Concluding Remarks

We have presented here a solution to the problem of providing fairness and security in a peer group that provides a number of improvements over prior work. In particular, configurable tolerance of masquerading, and the use of a trust model to assign privileges in a discrete, granular manner. We are also able to assure the fair participation of all nodes in the network, and expel members that act unfairly. This limits the damage that can be done by dishonest and malicious nodes.

Because we are able to assign privileges with greater granularity than previous models, we are able to trust nodes in the network only so far as their continued good behavior warrants. By continuously observing this behavior over time, we are able to increase our confidence in the continued honest and fair participation of all nodes. Should nodes choose to act unfairly or dishonestly, our expiration mechanism allows us to limit these disadvantageous behaviors to some time period. Because our model calls for these parameters to be defined for each group, our model can be adapted for use in a number of environments.

At present, our implementation has been validated, but is constrained to working in groups with small thresholds. In its current state, our implementation is clearly not suited

to running at large group sizes, or to be more specific, for the kinds of large thresholds which would be made use of by such groups. We have presented a number of suggestions both as to why this problem exists. In particular, we believe that the principle performance constraints are due to the sub-optimality of partial signature issuance, and to our use of locking mechanisms.

Looking forward, two important changes should be made toward the future of this research. First, our use of locking mechanisms should be investigated as the chief source of delay in packet processing, and subsequently as the greatest contributor to delay in the GMC acquisition process. Second, the partial signature issuance algorithm should be investigated, and reduced in the manner that we describe in § 5.3.

Furthermore, during our research, a number of problems with the underlying toolkit were identified, as described in this thesis. Solutions to these would provide both security and performance improvements. Moreover, during the course of our implementation, a number of improvements to the toolkit were made which are important to its security and robustness. These changes should be examined during future implementations, and integrated.

Since our work was completed, a more recent version of the Bouncer Toolkit, version `0.8`, has been released. Importantly, the limitations of version `0.7` of the toolkit which we describe should be examined in light of this new release. Changes we have made may benefit this version of the toolkit as well, and moving forward this should be examined.

Ultimately, our results contradicted our hypothesis that the implementation would run in groups with much larger thresholds. We believe that the reasons for this, however, have been satisfactorily resolved, and that the suggested solutions will enable the implementation to run as originally anticipated. For smaller thresholds, our work is able to provide the security mechanisms we have discussed at the cost of the overhead we have presented throughout § 5.

Given that our model is adaptable to a number of other situations, an important future contribution would be the application of the model to a second environment. At many points in our implementation the changes necessary to suit a different environment would be minimal. These changes can be made largely by constructing implementation-specific Monitor, Tracker, and Behavior modules. Particularly interesting would be results from an implementation with much larger or smaller re-certification requirements. The balance between delay in expulsion of non-honest members of the group, and performance cost due to re-certification could be explored in this way.

Appendix A

Use of the Implementation

This Appendix gives a brief outline of how to make use of our implementation. Source code is available from `http://wssrl.org`.

A.1 Installation

Refer to `INSTALL` in the source directory for detailed instructions on compilation, and configuration of the sources. Briefly, it should be done as follows.

```
./configure --prefix=/install_dir
make
make install
```

Now, the install_dir directory has been created, and is used for the remainder of the use of the programs.

A.2 Configuration

Once installed, the setup program may be run. The setup script requires the `dialog` package be installed [4].

```
sh setup.sh
```

The user is prompted for all necessary values required to create and run a test group. Refer to this document for a deeper understanding of what each of the values means. In general, the defaults may be used.

In addition to `setup.sh`, there is a second way to configure and use test groups. A number of scripts titled `5.sh`, `10.sh`, and so on are also present in `install_dir`. These scripts set a number of environment variables, and then call `setup.sh` which constructs the desired group automatically. The user may examine the contents of these scripts to determine how to construct groups in this way.

A.3 Bootstrap

Once a test group is constructed, it is ready to be run. First the two daemon processes are executed. The GA daemon is executed and passed a script generated by `setup.sh` as input. Each must be executed in the directory in which the binaries are found. `GNU Screen` [5] makes this process somewhat easier.

```
cd ca && ./cad
cd ga && ./gad < ga.config
```

Now, between the selected time periods $T_0 \rightarrow T_1$, the Trusted Domain is bootstrapped

by starting a number of nodes. The number of nodes that will bootstrap the TD is selected in `setup.sh`. Each is started in turn.

```
cd test/001 && ./servant
cd test/002 && ./servant
cd test/003 && ./servant
```

A.4 Remaining Members

Now, the user may choose to do one of two things. Either the Trusted Domain has reached the selected size, and the user may wait until T_1 to start the remaining nodes, or the remaining nodes may be started, and they will delay execution until T_1 has elapsed. The Trusted Domain must be started with the selected number of nodes, otherwise the TD dissolves.

New servants may be started until each node in the peer group is running. Nodes participate honestly by default. In order to exhibit non-honest behavior, one of the many misbehaviors must be selected. This brings us to command-line options for the `servant` program.

A.5 Malicious and Greedy Behavior

The first command-line argument to servant is a debug option that enables a delay when starting certain sub-processes. This value must be set to 0 to disable it unless debugging is desired. The second argument determines behavior type. Possible values are as seen in Table A.1. The third argument determines exactly which misbehavior the servant is to

run. Refer to Tables 4.4 and 4.5 for details on Greedy and Malicious behaviors, and refer to § 4.2.3 for details on Masquerading behavior.

Behavior Type	Value
Honest	0
Malicious	1
Greedy	2
Masquerade	3

Table A.1: Selecting `servant` Behaviors

As an example, to run Malicious behavior 1 (see Table 4.4) through `servant 4`, we invoke it as follows.

```
cd test/004 && ./servant 0 1 1
```

Note that some of the behaviors are listed as occurring *during* $T_0 \rightarrow T_1$. In this case a `servant` that is participating in TD formation must run this behavior, as opposed to trying to elicit the behavior *after* formation.

A.6 Masquerading

Masquerading behavior is elicited similar to the above, but with a slight difference. Since there is only a single masquerading behavior (to masquerade), the last argument to `servant` in this case is the virtual bandwidth (see § 3.2.12) to be assigned. To cause servants 5 and 6 to masquerade, with `servant` 5 having bandwidth of 60 bytes/sec and `servant` 6 having 25 bytes/sec, we execute the following.

```
cd test/005 && ./servant 0 3 60
cd test/006 && ./servant 0 3 25
```

A.7 Bringing it Down

The easiest way to take down the entire peer network once it has run for a sufficient period of time, if it is running locally, is to send all `servant`, `gad`, and `cad` processes the `TERM` signal, which each process handles by cleanly shutting down. This can be accomplished with

```
killall cad gad servant
```

For further details, please see the source code.

A.8 Miscellaneous Details

The Subversion [10] version control system was used to manage changes during this project. Thanks to Ben Collins-Sussman, Brian W. Fitzpatrick and C. Michael Pilato, authors of *Version Control with Subversion* [17].

Bibliography

[1] Bouncer Toolkit. http://sconce.ics.uci.edu/gac/download.html.

[2] Bouncer Toolkit Version 0.5 Source Code. http://sconce.ics.uci.edu/gac/docs/gac-0.5.0.tar.gz.

[3] Bouncer Toolkit Version 0.7 Source Code. http://sconce.ics.uci.edu/gac/docs/gac-0.7.0.tar.gz.

[4] dialog. http://hightek.org/dialog/.

[5] GNU Screen. http://www.gnu.org/software/screen/.

[6] Peer Group Admission Control Project. http://sconce.ics.uci.edu/gac.

[7] SGnut 0.1.2 Source Code. http://sconce.ics.uci.edu/gac/docs/sgnut-0.1.2.tar.gz.

[8] The Gnutella Protocol Version 0.6 from Gnutella Protocol Development. http://www.the-gdf.org.

[9] The OpenSSL Open Source SSL/TLS Toolkit. http://www.openssl.org/source/.

[10] The Subversion Version Control System. http://subversion.tigris.org/.

[11] Information technology – Open Systems Interconnection – The Directory: Selected Attribute Types. ITU-T Recommendation X.520 (1993 E), June 1993. ISO/IEC 9594–6.

[12] Information Technology – Open Systems Interconnection – The Directory: Authentication Framework. ITU-T Recommendation X.509 (1997 E), June 1997. ISO/IEC 9594–8.

[13] IEEE 802.11 Specification. ANSI/IEEE Std 802.11, 1999 Edition (R2003), 1999. ISO/IEC 8802-11: 1999(E).

[14] Abstract Syntax Notation One (ASN.1) Specification of Basic Notation. ITU-T Recommendation X.680, July 2002.

[15] ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). ITU-T Recommendation X.690, July 2002.

[16] Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/2004/REC-xml-20040204/, February 2004.

[17] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control With Subversion*. O'Reilly, 2004.

[18] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO '95: Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, pages 339–352, London, UK, 1995. Springer-Verlag.

[19] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3280 (Proposed Standard), April 2002.

[20] Yongdae Kim, Daniele Mazzocchi, and Gene Tsudik. Admission control in peer groups. In *NCA '03: Proceedings of the Second IEEE International Symposium on Network Computing and Applications*, page 131, Washington, DC, USA, 2003. IEEE Computer Society.

[21] Jiejun Kong, Petros Zerfos, Haiyun Luo, Songwu Lu, and Lixia Zhang. Providing robust and ubiquitous security support for mobile ad hoc networks. In *ICNP '01: Proceedings of the Ninth International Conference on Network Protocols (ICNP'01)*, page 251, Washington, DC, USA, 2001. IEEE Computer Society.

[22] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[23] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures. In *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 245–254, New York, NY, USA, 2001. ACM Press.

[24] Lik Mui. *Computational Models of Trust and Reputation: Agents, Evolutionary Games, and Social Networks*. PhD thesis, Massachusetts Institute of Technology, December 2002.

[25] Maithili Narasimha, Gene Tsudik, and Jeong Hyun Yi. On the utility of distributed cryptography in p2p and manets: The case of membership control. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, page 336, Washington, DC, USA, 2003. IEEE Computer Society.

[26] Paul Resnick, Ko Kuwabara, Richard Zeckhauser, and Eric Friedman. Reputation systems. *Commun. ACM*, 43(12):45–48, 2000.

[27] Nitesh Saxena, Gene Tsudik, and Jeong Hyun Yi. Access control in ad hoc groups. In *HOT-P2P '04: Proceedings of the 2004 International Workshop on Hot Topics in Peer-to-Peer Systems (HOT-P2P'04)*, pages 2–7, Washington, DC, USA, 2004. IEEE Computer Society.

[28] J. Schaad. Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF). RFC 4211 (Proposed Standard), September 2005.

[29] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[30] James Surowiecki. *The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies and Nations*. Doubleday, May 2004. ISBN:0385503865.

[31] Hao Yang, Xiaoqiao Meng, and Songwu Lu. Self-organized network-layer security in mobile ad hoc networks. In *WiSE '02: Proceedings of the 3rd ACM workshop on Wireless security*, pages 11–20, New York, NY, USA, 2002. ACM Press.