2005

Security in serverless network environments

Carl Holtje

Security in Serverless Network Environments

Carl Holtje
holtje@freeside.dnsalias.org

Department of Computer Science
Rochester Institute of Technology
Rochester, New York
United States of America
2004

Advisory Committee

Professor Hans-Peter Bischof, Chairperson

Professor Edith Hemaspaandra, Reader

Professor Phil White, Observer

Dr. Hans-Peter Bischof,
Graduate Coordinator

Contents

List of Figures

List of Tables

Abstract

As portable computing devices grow in popularity, so does the need for secure communications. Lacking tethers, these devices are ideal for forming small proximal groups in an ad-hoc fashion in environments where no server or permanent services are available. Members of these groups communicate over a broadcast or multicast network interconnect, and rely upon each other to form a cohesive group. While generally small in size and short in lifetime, security is a critical aspect of these groups that has received much academic attention in recent years.

Much of the research focuses upon generating a common, group-wide private key suitable for encryption. This group key agreement utilizes keying technology that is very costly for small, limited-lifetime devices. Furthermore, key agreement provides no constructs for message authentication or integrity. Traditional systems require two keypairs to address both aspects of the secure group – one for encryption, the other for message validation.

This work investigates the appropriateness of using a shared keypair for both contributory group key agreement and message quality guarantees. A JCE-compliant key agreement and digital signature framework has been implemented and is presented, and discussed. Using elliptic curve-based keys, this is possible at no loss in security, and these keys are easily and quickly computable on smaller devices. Algorithms that are known for their cryptographic strength are leveraged in both encryption and digital signature applications. This technique provides a computationally-efficient key agreement scheme and digital signature framework, and a network-efficient key and signature distribution system. Perfect forward and backward security is maintained, and all members retain a current view of the group from a cryptographic perspective.

This thesis is the culmination of several quarters of research and work, all conducted at the Rochester Institute of Technology under the supervison of Dr. Hans-Peter Bischof between December 2002 and January 2004.

This thesis is completed as partial fulfillment of the requirements for a Masters Degree in Computer Science from the Rochester Institute of Technology.

Chapter 1

Introduction and Goals

Server-based network security is a well-defined and well-explored topic that enjoys widespread and frequent use in modern computing. Transparent connections are made at all computing levels. From the secure web browser session to IPSec sessions between hardware devices, security is a critical and frequently overlooked aspect of networking.

Mobile computing with laptop computers is, in general, able to extend some of the constructs and techniques shared by their desktop counterparts. This assumes, however, a wired medium for physical network transport.

Wireless computing, on the other hand, presents undeveloped and fertile ground for research and development of security techniques. Many approaches that apply in the wired world simply are not sufficient in a wireless network structure. Access control, session cryptography, and authentication are major issues that have received a great deal of attention in recent years, and will continue to be the subject of much effort. Wireless devices mimic radio transievers because they are broadcast-based stations, capable of receiving all signals strong enough to reach their antennae. It is this broadcasting that is at the crux of the security issues. Man-in-the-middle attacks are both possible and relatively easy to execute. Denial of Service attacks are simply impossible to defend against, as there is no medium access control. Through all this, the goal is generally to configure a secure connection between a portable device and an access point to a larger network. This is, in a general sense, comparable to the client-server model with a few extra caveats.

A subset of wireless computing is the serverless environment, where there are no access points and a network is defined simply by other nearby nodes. The focus of this thesis is the ad-hoc serverless network topography. This environment is best described through example. Suppose a group of corporate officials meet on a job site to discuss and plan future developments. As many job sites are remote, it must be assumed there are no servers to provide security services such as key generation or authentication

(services akin to Kerberos). Each official is equiped with a computing device, such as a PDA or laptop computer, and each device is network-ready. For security reasons, it is important to be sure all meaningful communications are *encrypted*, and each message is verified as both *unmodified* and *sent from whom it reports its sender to be*. Each of these small devices are battery powered. The ability to form a logical grouping of these devices, generate a shared key using influences from each member, and communicate through these devices securely is needed.

As with any system, there are constraints that this system as well as each member in the group is assumed to work within and abide by. In an ad-hoc wireless network environment, these constraints are:

- No central servers or services exist for security-related roles.
- Each device is within proximal broadcast range of every other device participating in the group.
- This broadcast medium is unreliable – message delivery or proper ordering is *not* guaranteed.
- Message routing is not supported.
- Each device works in a manner consistent with group security, and does not work to limit or inhibit the functionality or security of the group.
- Each device is able to compute the cryptographic components to facilitate group security in a timely manner.
  - It is permissible for larger systems, within proximal range, to participate in a group. This requires that employed techniques be sufficiently strong to not be compromized by any reasonable system (i.e., a desktop computer should not be able to crack any security constructs used in this environment).
- Membership is dynamic and somewhat volatile. Security constructs must support this instability to provide perfect forward and backward secrecy, as well as general key freshness.
- The responsibility of group maintenance must not fall upon a single member consistently. This encroaches too closely to the server/service paradigm, and reliance upon this service will prove detrimental to group security, stability, and scalability.

There are also a set of issues that this system will not address. These are as follows:

- Broadcast network communication has unreliable delivery; reliability issues are not addressed in this thesis.

- Each message sent must fit within one maximum transmission unit (MTU). Multi-MTU messages require some degree of reliability; this topic is addressed later in Section 8.4.
- Denial of Service attacks – these are impossible to defend against in this environment.
- User uthentication and Access control – this work defines the events immediately following user authentication and access control protocols.
- Group membership changes are announced – the current version does not handle silent member leave events (moving out of broadcast range, power failure, etc.). This would likely be handled by a heartbeat and heartbeat monitoring functionality [12].
- In terms of member operations, only single-member operations are supported in this version. Multi-party operations such as group merge and partition are not supported at this time.

  Conceptually the multi-party operations could be represented as a series of single-member operations. For scalability and stability reasons, this would likely result in a rewrite of a significant block of the protocol. This is because it is possible in situations of multi-party operations that no single member be able to compute the required subtree, and inter-subtree communication and agreements would be required to complete the full agreement sequence.

It is one of the goals of this document to provide an overview of the issues and current approaches to the issues surrounding serverless networking. Additionally, a novel approach to minimizing key computations for key agreement and message authentication/integrity validation is presented and discussed. Included in this discussion is a review and performance evaluation of the implemented framework leveraging elliptic curve cryptography and modern encryption techniques.

1.1 Motivation and Goals

This work serves as a subsystem to the Many To Many Invocation/Many To Many Protocol (M2MI/M2MP) framework as developed in the Computer Science Department at the Rochester Institute of Technology [13]. It is important to note, however, that this security module does not rely upon *any* functionality from the M2MI/M2MP framework. Compliance with the `edu.rit.m2mp.Channel` interface is maintained; some type-based dependance is afforded.

As development progressed in the M2MI/M2MP project, it was realized the issues surrounding security and making a broadcast- or multicast-based system secure were numerous. As a result development diverged from security issues, and its inclusion was left for later.

Security Goals

It is the goal of this thesis to provide a secure channel of broadcast communication, situated within the M2MP framework. None of this effort relies upon the larger framework in any functional way. This requires a series of distinct facets be addressed head-on:

1. *Confidentiality* – Knowing and being guaranteed that it is computationally infeasible for parties other than the intended receiver to determine the contents of a message. This is typically accomplished through the use of cryptography, as the goals of cryptography are privacy of data.

   In two-party communications a series of options exist. For symmetric cipher systems, a shared common key is fed into an encryption scheme. This key is agreed upon using a key agreement protocol, much like Diffie-Hellman. For asymmetric cipher systems, a set of keypairs must exist and be known. A sender would then encrypt a message with the receiver's public key, and the receiver would then decrypt with their private key. This requires no key agreement, but does require an up-to-date key database that must be known and trusted to be secure and safe from tampering.

   In multi-party communications, as in collaborative broadcast-based groups, encryption serves as a membership border. By the nature of broadcast, all nodes within a sender's broadcast range receive the message, but with encryption, only those nodes with knowledge of the encryption key will be able to understand the contents.

2. *Group Key Agreement* – Multiple network nodes participating in a common group, sharing a common cryptographic key. This is an area of active research as restrictions on both computational complexity and network utilization are frequently imposed. It must be enforced that perfect backward and forward key secrecy be maintained to prevent any decryption of any messages received while not a member of the group.

3. *Integrity Verification* – Knowing and being guaranteed that a received message has not been tampered with or altered in any way. This is generally done with the employment of hashing. A message of n length is represented by a constant- length

bit string of k bits. Any change in the message results in a dramatically different hashcode, and any alteration of the hash will clearly not match the data. Using just hash code values does not prevent a third party from modifying the payload and injecting a new and valid code.

4. *Authentication* – Knowing and being guaranteed that a received message from sender S was in fact sent by S. This functionality is combined with integrity verification, and provided through the use of digital signatures, and, in a more connected world, supported by digital certificates and the Public Key Infrastructure (PKI).

   Signatures are based on public and private keypairs. A message would be signed with the sender's private key, and later verifed with the sender's public key. This provides irrefutable proof a particular sender sent any given message. Signing includes a mixture of private key and a hash of the message. This verifcation may be done by any party privy to the signature and the signers public key.

The use of elliptic curve-based keys will be evaluated for appropriateness in group key agreement and digital signature. A protocol using objects and object serialization/deserialization will be implemented supporting member join and leave group events. Key-related events and calculations will be compared to RSA/DSA style keys, as generated by the Java security framework (formally known as the Java Cryptography Extension, or the JCE).

Constructs based on RSA/DSA-style parameters will be compared and contrasted with curve-based security systems, and evaluated in general terms for their appropriatness in group settings.

Part I

Background

Chapter 2

Current Security Constructs

Complete network security is comprised of three distinct facets – confidentiality, message integrity, and message authentication. Confidentiality can be gained through encryption, following a sequence of one or more key agreement rounds. Message integrity and authentication are ensured through the use of digital signatures.

2.1 Key Agreement

Several protocols have been defined for this purpose, such as CLIQUES [35], Tree-based Group Diffie-Hellman (TGDH) [23], [24], Logical Key Hierarchy (LKH) and their extensions (LKH++) [30], and selected others.

There are, in general, two schools on group keying. One, the *distributory* school, relies on a single member to exist in all subgroups and survive from the the beginning to the end of the group's lifecycle. It would be the responsibility of this member to construct and deliver the group key to all group members. While this is very simple, it violates a critical set of our intentions (most notably reliance upon a single group member for providing cryptographic data to the group). The second group keying philosophy, the *contributory* philosophy, is the de-facto technique. This technique requires every group member influence the final key in some way, and relies upon no constant or common services or members.

Other secure group environments rely upon every node owning a copy of the key prior to membership; not quite the distributory or contributory designs. This environment would then simply require access control constructs, and provides little security to group members. This works if all members and potential members have the same priviledge to access the network and guarantees can be made that those elements that should not be members cannot be members.

Regardless of which key agreement style is chosen, possession of the correct group

key is used to delineate group boundaries; those nodes with the correct key are in a specific group, while those without are not.

Below is a discussion and an evaluation of the most common techniques of executing group key agreement.

All key agreement techniques implement some form of the Diffie-Hellman key agreement algorithm, discussed further in Section 6.3.

2.1.1 Linear Group Key Agreement

Linear group key agreement techniques such as CLIQUES [35] provide a simple path to key agreement for a group of devices.

In this scheme, all group members must obtain the entire key string (that is, all intermediate agreement key sequences). Each node computes the final key based on the values of every other partial key in the group. This completes in $O(n)$ time.

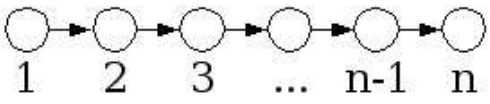

Figure 2.1: Linear Key Agreement

Figure 2.1 shows the truly linear nature of this style of key agreement. Supposing we start at node 1, our private key would be used to generate a common shared key with node 2, which would then be used in an agreement with node 3. This process would continue for all n nodes. Any other node would also start at with node 1, but clearly skip their own public key value.

All nodes visit all other nodes, and compute similar intermediate keys (these will clearly not be identical as the initial private key differs), generating a common final group key.

There are two popular techniques to approach linear key agreement. One places the burden of computing the new key upon the joining member, the other upon an existing member.

The first technique requires the new member receive all existing public keys from the group, append their public key, and fully compute the new group key. A broadcast message will deliver this new key chain to the group. This is secure because the current group key (pre-join) has been computed without inclusion of the joining member; he's gained nothing from the group and the group has lost no security. The joining member can compute a new key using his private key and the group's public keys, but this key will be very different from the existing group key.

The second technique, as utilized by CLIQUES [35], passes the entire key chain to the joining member. They then append their public share, modify (via intermediate key agreement) every other keyshare in the chain, and pass the new chain back to their joining sponsor. The sponsor then computes a new chain of intermediate key values using their private key, and distributes this final chain to the group. Each node can then locate their subkey, and recompute the final group key. This technique is clearly more intensive from a resource allocation and computational perspective, and as such, generally not desireable for implementation.

Issues surrounding linear key agreement in general are further discussed in Section 8.2.

2.1.2 Tree-Based Group Key Agreement

Tree structures provide a method of key agreement and management that has several desirable qualities. The very nature of being in a tree formation reduces the complexity from $O(n)$ to $O(\log n)$ and provides a well-suited structure for dynamic membership changes. As elements change membership status, the tree requires only a few key-node updates to modify the whole tree. This means only a few messages are passed, reducing computational complexity and bandwidth.

Logical Key Hierarchy (LKH)

This scheme, as presented by [30], places all group members at the leaves of an order-m B-Tree, and consists of a series of intermediate subgroup keys. Built from the bottom up, intermediate keys higher in the tree serve greater numbers of group members, where the root of the tree is the group key.

The keys delivered to each member are shared, in essence, with no other nodes with the exception of the root. All intermediate keys, however, are shared by all dependents (children in this subtree) of the inner-tree node.

This group tree, in most LKH implementations, is stored on one designated host. It is clear from the role that this node must be a nearly permanent fixture in the model, as failure would place the group in a state of near total confusion. This could be tempered by distibuting the entire key tree to a few network elements, thereby permitting group recovery from a known state, and prevent total group reconstruction.

Membership changes initially affect only those keys above and to the most direct path to the root. As these intermediate values change, a secondary flow will change all other members keys to guarantee perfect forward security and key freshness.

This minimal-impact effect of trees initiated by membership changes helps key distribution problems immensely.

Some thought about this design will reveal an interesting observation – LKH is a combination of linear- and tree-based key agreement. On one hand we have the tree structure maintaining the tree, but to do an agreement at a given tree node, a linear sequence of agreement rounds needs to occur. We gain very little computationally or in network terms using LKH over linear agreement techniques.

Figure 2.2 shows the general form of LKH agreement on an order-k tree. The first leaf node, for example, is network elements 1 through $(k-1)$.

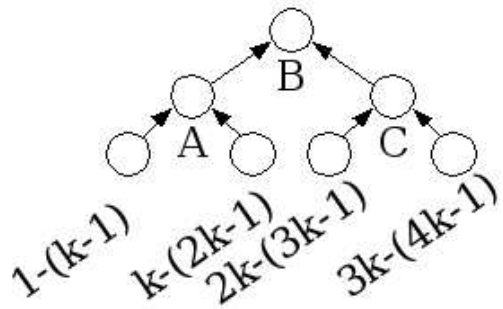

Figure 2.2: Logical Key Heirarchy Key Agreement

Extentions to Logical Key Hierarchy (LKH++)

The paper by DiPietro et al ([30]) describes an extension to Logical Key Hierarchy that uses hashing and intermediate key values. Using a key distribution node, dubbed the *center*, this protocol attempts to require a minimal amount of network communication and message computation to generate and share the group key. As in other techniques, intermediate values between any given node and the root are required to be maintained by each node.

The main difference between this protocol and other protocols is the use of hashing. The authors of [30] describe how a new key, K' is to be sent to the group, encrypted with the current group key K. The center computes the one-way hash of K (written as $H(K)$), followed by the xor of the new key and the hash of the current ($\hat{K} = H(K) \oplus K'$, and sends this to the group. Group members then recover the new key by computing $K' = H(K) \oplus \hat{K}$. By hashing the values, the key is never sent in the clear or even as a collection of partial values. Rather, its delivered in a pseudo-encrypted form that only current members, those elements with the current key, can decode and retrieve the new key.

The benefits of this technique are a general reduction in resource consumption. They, [30], noted a fifty percent drop in computations done by the key center, and a fifty percent reduction in required bandwidth for group formation.

Doubly-Rooted Trees

As an extension of basic singly-rooted trees, Diamonds [32] defines a tree structure that has many desireable properties. Firstly, as a means to maintain group functionality with basic survivability properties, each node is bi-connected. The simplest way to achieve this is by arranging all nodes in a circle. This has a side affect of having a diameter that increases with each member addition, which results in increased network latency. Diamonds, on the other hand, are recursively defined as having two links to other nodes. This helps maintain the connectivity that a singly-connected graph (binary tree) can not have, and the structure grows logarithmically with increased membership.

Aided by structure maintenance functionality, the diamond graphs are easily maintained and restructured to maintain their bi-connected status. As the diamond is recursively defined, nodes are positioned in the structure in a manner preserving ballance. As one side of a diamond grows, so must the other to maintain this ballance. It is this ballance that allowed nearly horizontal growth patterns in performance tests [32] for group reconstruction computations, and slight growth in latency as group sizes grew from five to fifty.

Figure 2.3 shows the basic form of the diamond structure.

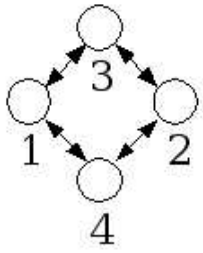

Figure 2.3: Double-Rooted Key Agreement

Full and Complete Binary Trees

The tree structure used in the implementation of this Thesis is very similar to the Logical Key Hierarchy system, with a few important deviations. While LKH is an order-m B-tree (a binary tree with instead of two children, up to m children are supported), our tree is strictly order-one (one key value with two child references). Instead of the tree being stored and maintained by one central node, the tree is distributed throughout the entire group. This aids in general group scalability as well as stability.

This key agreement structure was chosen for its simplicity both algorithmically and computationally.

Figure 2.4 shows the general form of the key agreement structure as implemented by

this work.

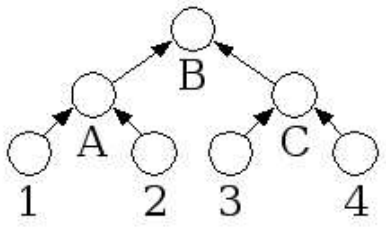

Figure 2.4: Binary Tree-Based Key Agreement

2.2 Message Authentication

In general, message integrity and authentication is achieved through digital signature techniques. Recent advances in encryption provide some authentication guarantees, but these are generally ineffectual for this setting (see [22], [21] and [33]). Such encryption schemes are useful between two parties where each can inject a unique sequence, the nonce or a counting value, and use it in an authentication capacity. In the multi-party setting, this would require a lookup table of member-to-sequence, and would incur futher message processing overhead. Additionally, this would impose greater processing requirements at the time of member-join and in key distribution.

Traditionally, keys used in key agreement are inappropriate for digital signature applications. While this is true for RSA/DSA-style keys, this is not the case for elliptic curve-based keys. This halving of group keys reduces the total number of keys that would otherwise need distribution to the group and generally simplifies the inner- and inter-node protocols.

Having a public and private keypair at the root of our key tree provides us with one additional feature. While not investigated in this work, a group signature would provide an inter-group security construct. Other work has investigated this further.

2.3 Message Authentication Concerns

Using digital signatures requires a keypair suitable for this purpose, requiring distribution of a second key for each node in this group. Requiring a second key to be generated, maintained, and distributed for each node serves as the motivation behind using the same key for key agreement and signing, as investigated by this thesis.

Digital signature keys are generally static entitities with a reasonably long lifetime. In this dynamic setting, however, long-term keys are both a vulnerability and a logistical stumbling block. They are a vulnerability in the sense that with each message that is

signed, a portion of the private key is revealed. Slowly, the entire private key could be revealed. Long-lifetime keys are a stumbling block as maintaining a database of trusted verification keys over a period of time becomes a storage issue and a practicability issue. It would be far easier to simply regenerate a key for each group than to try to maintain a history of all members keys in all previously attended groups. Additionally, this would require that every participant in a group maintian a secure and off-network storage of their signature keypair for restoration after battery replacement – no more battery hot-swap in the field. Using temporal signature keys would allow the group key to change over time and membership, without the storage or server-based constraints.

Part II

Thesis Work

The code phase of this thesis work included implementing several aspects of the secure broadcast channel. Included in this codebase is:

- JCE-compliant Digital Signature provider, implementing the HORS digital signature algorithm [31].
- Message hierarchy.
- Message serialization support.
- Protocol definition.
- JCE-compliant tree-based key agreement module for RSA/DSA and elliptic curve-based keys.
- JCE-compliant digital signature module using RSA/DSA and elliptic curve keys.
- JCE-compliant cryptography module using RSA/DSA and elliptic curve keys.
- Broadcast network communications point.
- Support classes to maintain state and references for framework.
- Adaptation of Java-internal classes for JCE key manipulation.
- Integration with M2MI/M2MP.
- JavaDoc comments of all components of all implemented elements.

One aspect of this thesis is to make encryption and message authentication more streamlined. This has been accomplished by using the same keys for both operations. This technique has not been explored in other security related research. An evaluation of this idea is provided in chapter 6.

The following chapters discuss the overall design of the security framework from a high-level perspective, and then describe individual components. A tour of both inner- and inter-node operations and proceses, and finally a cryptographic analysis of components is provided.

Chapter 3

System Design and Specification

The security framework is divided into a collection of modular blocks as follows:

Communication Network I/O in a non-blocking broadcast fashion.

Messages Messages are the method of communication. Designed and manipulated as Java objects, messages must be serialized before communication, and deserialized back into objects upon receipt. An object-oriented design makes handling and inner-node processing easier.

For efficient marshalling, these message objects all have common ancestry and follow a common template of design. This outlined further in Section A.1.

Key Agreement Core of security constructs. This provides storage of all member public keys as well as the functionality to generate common group key.

The tree-based system is modeled after work described in [24] and uses a common tree-based key storage system for agreement and maintaining digital signature verification keys. It is this framework that provides the private key values to encryption and public key values of other group members to the digital signature verification modules.

Signatures Module that optionally provides message authentication support through the use of digital signatures. Ties closely to the Key Agreement module for members' verification keys.

Cryptography Module that provides the optional cryptographic utilities to ensure group-wide secrecy.

Message Handling Tied closely to both the communication and message modules, the message handler is the core of the processing infrastructure. This is defined by a reactor-style event-driven sequence. Messages are processed in object form (see

the Messaging description above and below) and acted upon in the order in which they are received.

Provisions have been taken to ensure that user-level messages (`Application-Message`s) are processed simultaneously with all protocol-level messages that are exchanged. This provides a nearly transparent channel of secured communications.

These messages are, however, only processed upon achievment of Member status.

State Repository Classes whose sole responsibility is to maintain relationships to the various high-level modules and maintain protocol and group member state information.

M2MP Interface Written as an M2MP Channel implementation, this class is the only class that needs instantiation for incorporation into any M2MI/M2MP application.

3.1 Network

The design of the network interface for this project was very simple. In an attempt to acurately model a broadcast environment, like that of a wireless device, this channel utilizes the broadcast network. Messages are written to the IP address 255.255.255.255, and read from the address 0.0.0.0. This is correct for broadcast communication.

Some rate limiting is done to help control the amount of data being sent from any one node. This primarily serves to increase reception rates of messages as a high rate of dispatch was shown to steadily decrease receipt rates.

3.2 Message Processing

Each node receives all messages sent in the group, but only processes messages addressed to two values – those messages 'unicast' to their ID, and broadcast messages sent to the entire group.

All messages flow through a common processing sequence:

1. Data is received into a byte array by the `CommPort` network interface.
2. This byte array is passed to the `MessageHandler`.
3. A quick check is made to be sure the message is not from ourselves.
4. Message reconstruction follows, where these steps are taken:
   (a) A first attempt at message reconstruction is made. This is done via library call to `MessageUtils`.

(b) If this reconstruction fails, a second reconstruction is attempted after decryption with the current key (assuming encryption is enabled).

(c) If this second attempt also fails, the message is considered unrecoverable and simply discarded.

5. If a message object has been recovered, we first check to ensure the message is not from us, and then for a digital signature. If a signature is present and signature verification fails, this message is discarded.

   While this may seem extreme, it is understood that if it was important enough to sign, it must be important enough to successfully verify.

   Accepting a message that fails signature validation might almost not have been signed.

6. An exising message is then processed based on message type and both node and protocol states.

7. During processing, the incident message is consumed. A reply message may be generated (depending on the state and behaviour of the protocol).

   (a) If signatures are enabled, this message would then be digitally signed.

   (b) Likewise, if cryptography is enabled, this message would then be enciphered with the current group key.

   (c) The final byte array of the processed message is then sent via the `CommPort` instance.

3.3 Security

Security in this framework is broken into two phases – privacy and message authentication. Privacy is supported through encryption, with authentication through digital signatures.

3.3.1 Key Agreement

Using a full binary tree (a tree where each element has either zero or two children), we can efficiently compute a group key in $O(\lceil \log n \rceil)$ agreement cycles.

Two important observations about groups and group security must be highlighted. The first is that smaller devices cannot compare computationally to larger computing devices. As a result, these smaller devices run a risk of cryptographic compromise. The nature of signing a message is a partial revealing of the sending party's private key. With time, the entire private key is revealed, and a patient and determined third party could begin forging valid message signatures.

The second observation is that the keys and signatures used in this setting are very short-lived. Public keys used in key agreement and signature verification are valid only as long as the lifetime of the group or the duration of a members' affiliation with the group.

Combining these two observations motivates the use of dynamic, temporal signature keys, as described by [31, 10].

3.3.2 Message Integrity and Authentication

Message integrity in the strict sense is ensuring data received has not been modified. A naïve approach would be simple checksums, but a fatal pitfall to this technique is that a third party may modify the data and insert a modified, and correct, integrity checksum. Clearly this approach will not suffice.

Message Authentication Codes (MAC's), or more specifically Hashed-MAC's (HMAC's) are also inappropriate for this environment. These constructs provide data integrity validation, but do not provide message authentication, as the authenticating bytes are generated and verified using the same cryptographic key. Message authentication codes are useful for quickly determining if a file on a computer has changed, but little more [25].

Traditional algorithms for message authentication rely on digital signatures, generally accompanied by a digital certificate chain to prove authenticity. Digital signatures provide the same unique hash value as MAC's, but include verifiable data from a private key value. This allows a receiver (who, it must be assumed, possesses the associated public key) to verify both the identity of the sender and the message data.

While the dynamic network system will also utilize digital signatures, one critical difference exists with a more connected computing paradigm. In this setting, there is no way of verifying a chain of trust as presented in a certificate. This means, technically, we really do not know who the sender of a message is, but we do know that the message was not modified in transport. We can associate a message with a device or a user, but have very little proof of user authenticity.

Additionally, signature and key agreement keys are separate entities. This is done for algorithmic and mathematical integrity rather than security purposes. As discussed in Section 6, this thesis uses elliptic curve-based keys which are able to work for both applications.

The unique strength of elliptic curve keys is presented in this thesis – using the same keys for generating the common group encryption key as for signing messages. The private key is used to sign a message, whereas the public key, distributed as part of the key tree to the entire group, is used for signature verification. Other research does not

explore this application of dual-role keys.

This technique provides us with several desireable affects. First, we have one key to distribute per node instead of one key for encryption key generation and one key for signature verification. Second, when a node serves as group sponsor for a joining or leaving member, their keypair is regenerated. This enforces both forward and backward secrecy from an encryption point of view, but also updates the signature keys – our dynamic and temporal signature system.

Chapter 4

Completed Work

The security framework consists of a series of packages representing functionality. These are listed and discussed here.

Many of the cornerstone classes (eg, the `*Engine` classes) have a `report()` method to report statistics and current state. This is useful for monitoring the behavior and condition of the various modules, and is accessable via a method call to `report()` from an instance of the `SecureChannel` class.

Section A.2 provides a high-level view of module interaction.

Package listing and functionality description:

The `FlexiECProvider` Provider Package Provides all the elliptic curve-based functionality as used by this research, including key generation, key agreement, and digital signature functionality. Implemented as JCE provider, the implementation is compatable with several international security standards. Available from `http://www.flexiprovider.de` [16].

The `FlexiCoreProvider` Provider Package Also a JCE Provider, this package provides curve-compatable encryption routines, secure hash functions and secured random number generation. Available from `http://www.flexiprovider.de` [15].

`edu.rit.m2mp.security` Top-level package contains `edu.rit.m2mp.Channel` implementation via `SecureChannel` class. All other classes in this framework are utilized by the channel and should not be instantiated by other classes or applications.

`edu.rit.m2mp.security.comm` Provides non-blocking broadcast network communication support. Using a thread pool from the `edu.rit.m2mp.security.utils` package, multiple threads are capable of reading and processing in quick succession to permit prompt reaction to incident messages.

Unlike their wired counterparts, there is no concept of medium access control in a broadcast environment. To this end, we cannot assume a single message being

available for network reading at any one time. This scalability supports greater group stability and consistency.

edu.rit.m2mp.security.crypto Provides JCE-based encryption through all currently-loaded JCE providers. The Sun provider supplies several encryption algorithms sufficient for the scope of this work.

edu.rit.m2mp.security.handlers At present, this package contains the 'pinging' thread used in group formation/discovery.

edu.rit.m2mp.security.keyagree Provides tree-based key agreement functionality with the sole purpose of group key generation.

edu.rit.m2mp.security.messages Defines message framework and hierarchy. This is diagramed in Section A.1. Also provided is the functionality to decompose and recompose message objects to and from serialized byte arrays.

edu.rit.m2mp.security.signatures Provides JCE-based digital signatures for messages. Tied closely to the key agreement classes through a `KeyProxy` class, the public shares of members' public-private keypairs are used for signature validation.

edu.rit.m2mp.security.signatures.hors.* Implementation of the Hash to Obtain Random Signatures (HORS) digital signature algorithm as described by [31]. This is not used by this system as public keys are too large to be practical. This remains in the source tree as proof-of-concept work.

edu.rit.m2mp.security.utils Utility classes to support number translations to and from byte arrays and array-based operations. Also home to the `ThreadPool` class used by the `CommPort` class for multi-threaded reading.

***(several)*.test** Several packages have test sub-packages to examine functional performance in their parent packages.

4.1 edu.rit.m2mp.security

"Root-level" package that provides a coordination point and access point into the security framework.

SecureChannel Implementation of the `edu.rit.m2mp.Channel` interface, and is the only class that should be instantiated by any application. This class provides a

foothold into the security framework, providing status, accessor and mutator methods for aspects controlling inner operation. The ability to redirect error messages and status messages to other displays is also possible.

MessageHandler Reactory-style message processor of system. Receives messages from edu.rit.m2mp.security.comm.CommPort, decrypts byte array, reconstructs object, verifies signature, and processes message. If message results in state change or requires reply, reply is sent to edu.rit.m2mp.security.comm.CommPort for transmission.

ProtocolConstants and Repository Maintain state and inter-module references for inner-framework use. The ProtocolConstants class is a collection of constant values for configuration of options and general operation.

ExceptionHandler and ExceptionListener Provide a means to redirect error messsages and exception handling. Many exceptions are passed up through this system to the Channel. This provides the ability to deliver security-layer messages to a user for display or extended handling. By default, these are written to standard error.

Each of these classes are interfaces to allow additional implementation. The MessageHandler class implements these interfaces, and uses the DefaultExceptionHandler and DefaultExceptionListener classes to write to the standard error stream. The NullExceptionListener consumes all messages and displays no output.

OutputHandler and OutputListener Similar to the ExceptionHandler / ExceptionListener pair, this provides a redirection of status and reporting messages. This provides a means of obtaining system-wide status report information for user display or inquiry by other systems.

Each of these classes are interfaces to allow additional implementation. The MessageHandler class implements these interfaces, and uses the DefaultOutputHandler and DefaultOutputListener classes to write to the standard output stream. The NullOutputListener consumes all messages and displays no output.

4.2 edu.rit.m2mp.security.comm

Package that defines a network-knowledgeable endpoint for arriving and departing messages. These are written as broadcast UDP packets.

CommPort Network-level access point. The current implementation reads and writes to the broadcast address.

A ready set of threads is kept warm for prompt reads from the network channel. This is provided by an instance of the `edu.rit.m2mp.security.utils.ThreadPool` class.

Reader Small object to read from network. This was broken out of the `CommPort` object to allow higher-speed multi-threaded reads.

Minimal effort would need be expended to convert this output device to other types of connections. It may also be desired to convert this system to a pipe-like implementation of the `edu.rit.m2mp.Channel` interface instead of an endpoint design.

4.3 edu.rit.m2mp.security.crypto

CryptoEngine Provides access to JCE and other cryptographic functionality. This allows selection of many types of encryption algorithms useful in testing.

Cryptographic algorithm is dependent upon key style currently in use; elliptic curve keys do not work in the JCE ciphers expecting RSA/DSA-style keys.

Two instances of the `javax.crypto.Cipher` class are used; one is initialized for encryption only, the other for decryption only. This provides the fastest possible processing time as two messages could be encrypted and decrypted simultaneously without synchronization issues.

4.4 edu.rit.m2mp.security.handlers

Discovery Small class to do pre-membership probes. This functionality runs in a thread that is only terminated when the desire to not be in *any* group is registered with the Channel. This allows automatic rediscovery of a group in the event of disconnection.

Once the protocol for membership has begun, this thread is assigned a low priority. This permits state-probing for re-activation, but keeps resource consumption minimal.

This could be duplicated and modified to perform group-to-group discovery with the goal of forming the largest possible group. Scalability issues arise as group rekeying messages can become too large for practicality.

4.5 edu.rit.m2mp.security.keyagree

Package designed to maintain the classes required to support tree-based key agreement.

AgreementEngine Provides the front-end to key agreement, parameter generation, key operations including key generation, regeneration from bytes, and agreement. Also provides high-level functionality for member operations such as member join and member leave.

AgreementImpl Parent class to all agreement techniques. This provides common functionality used in many agreement styles.

TreeAgreement Full implementation of tree-based key agreement and key-based operations (eg. key refresh). Extension of the `AgreementImpl` class.

Tree Maintains the current view of all members' public keys and intermediate public/private key values. This is based primarily on descriptions of work done in [24] with optimizations and enhancements implemented where needed.

Node and MemberNode Containment and organizational classes used within the tree to maintain member data. Instances of `MemberNode` represent group members and exist at the leaves of the tree, where the remainder of the tree consists of `Node` objects.

KeyProxy Provides other packages, specifically the `edu.rit.m2mp.security.signatures` classes, access to all public keys currently active in the group. It is through this class that key queries are funneled.

4.6 edu.rit.m2mp.security.messages

Definition of objects used to control inter-node security constructs and protocols. These messages never extend into application-space as they are entirely consumed by the channel constructs.

The format of all messages is as diagrammed in Figure 4.1. It is this format that allows robust and prompt serialization and deserialization of message objects during transport phases.

It is also important to note that not all messages carry an explicit payload; some messages relay information and intent by their type, for example the `LeaveRequest` and `AgreeACK` messages.

GroupMessage Interface that defines the basic operations supported by all messages in the protocol.

Start Byte	Byte Count	Optional	Description
0	1	no	Protocol message identifier
1	1	no	Message type identifier
2	4	no	Source node Id value
6	4	no	Destination node Id value
10	4	no	Message sequence number
14	1	no	Digital signature length
15	≥ 0	yes	Digital signature (depends on algorithm)
≥ 15	1	no	Payload length
≥ 16	varies	yes	Payload

Table 4.1: Message Packet Structure

GroupMessageImpl Provides the base implementation of many of the methods defined in the `GroupMessage` interface for inheritance by the various message classes. All message objects in this package must extend this class for consistent functionality.

MessageConstants Storage of message-oriented constant values; used by `MessageUtils` and `MessageHandler` (from the `edu.rit.m2mp.security.security` package) classes for various processing routines.

MessageUtils Provides a common serialization and deserialization framework for messages.

4.7 `edu.rit.m2mp.security.signatures`

Parent-package supplying digital signature functionality. Both JCE and elliptic-curve algorithms are available, but their use is predicated on which key algorithm is currently in use.

Just as the encryption module uses two service providers, this module employs two instances of the `java.security.Signature` class. Again, this allows simultaneous processing of two messages without suffering from synchronization issues.

SigningEngine Tied with the key agreement package via a `KeyProxy` object, manages digital signature signing and verification processes.

When signing a message, the following message fields are included in the signature hash:

1. Message type

2. Message source identifiers
3. Message destination identifiers
4. Message sequence number
5. Message payload

For enhanced response and processing time, initialization of the signature module includes two separate signature objects; one for signing, one for verification. This limits the requirement for behavior mode switching with every message.

As mentioned in the discussion of the `KeyAgreement` package, the members' public keys used in signature validation are accessed by an instance of the `KeyProxy` class. This limits key data replication within the security framework.

4.8 edu.rit.m2mp.security.signatures.hors

The HORS algorithm (Hash to Obtain Random Signature), as described in [31], was implemented as closely to the specification as possible.

This was implemented as a JCE-compatable Provider object.

While functional, this technique is not used in the greater framework for performance reasons – public keys and signatures were too large for general use in this small-message environment.

A partial solution is to hash the final signature; this will provide a collision-resistant fixed-length representation of the signature. This improvement will not, however, affect public key sizes.

Additionally, these keys could not be used for group-wide key agreement without additional modification and security analysis. This inadequacy violates the goals of using a common key for both signatures and encryption-key computation.

A insiders view of a HORS keypair and key serialization/deserialization is available in Appendix C.

HORSProvider Gateway class used by Java Security framework for referencing and loading requested functionality.

HORSParameterGenerator Generates random values used by HORS algorithm during key generation.

HORSKeyPairGenerator Using parameters and random seeds, generates a keypair to be used in digital signature generation and verification.

HORSKeyFactory Reconstucts an encoded key into key objects.

`HORSPrivateKey` and `HORSPublicKey` Private and public keyshare values for signature keys using the HORS algorithm. As with all other digital signature algorithms, the private key is used for signing data and the public key is used for signature verification.

The public key is a function of the private, and should provide no easy way to reconstruct the private key (ie, using a private exponent or multiplier).

`HORSSignature` Main functionality class that processes all signing operations.

The HORS algorithm, like other cryptographic systems, has three distinct modes of operation - parameter generation, signing of data, and verification of this signature.

Parameter Generation

```
int l, k, v, h
while (true) {
  l = randomValue()
  if (l >= 128 && l <= 768)
    if (l mod 64 == 0)
      break
}
return quad[l, 16, 160, SHAIdentifier]
```

The values of $k = 16$, $t = 160$ and using SHA for hashing are default and set for testing and compatability. Other values for l, k, and t were suggested and evaluated in [31].

The bit-size of l may be passed as a parameter, as well as the desired hashing algorithm to influence parameter generation.

Signing

Assume integer parameters t are k established.

```
byte[] data
PrivateKey key

byte[] hash = oneway(data)
int[] substrings = new int[k]
substrings = breakHashIntoSubstringsModuloT(hash, substrings, t)
byte[] signature = new byte[0]

for i = 0 to k
  signature += key[substrings[i]]

return signature
```

Verifying

Assume integer parameters t are k established.

```
byte[] sig
PublicKey key

byte[] hash = oneway(data)
int[] substrings = new int[k]
substrings = breakHashIntoSubstringsModuloT(hash, substrings, t)
int[] blocks = new int[k]
blocks = breakSignatureIntoSubstrings(sig)

for i = 0 to k {
  if (key[substrings[i]] != oneway(blocks[i]))
    return false
}
return true
```

The `oneWayHash` function generates a byte array representative of the parameter data. This implementation supports SHA-based hashing as well as a custom implementation of a hashing algorithm (algorithm selection is configurable).

This custom hashing algorithm is loosely based on the `hashCode()` method of the `java.math.BigInteger` class, and like SHA, returns a constant-length hashcode.

The `breakHashIntoSubstringsModuloT` method segments this known-length hash and segments it into k equal-length substrings. Each of these substrings is stored as an integer value modulo t.

4.9 edu.rit.m2mp.security.signatures.hors.interfaces

HORSKey, HORSPrivateKey, and HORSPublicKey Define the basic functionalty and provide type safety for keys in the HORS framework.

HORSKeyPairGenerator Define the basic functionality for a key pair generator, as well as comply with required JCE patterns for adaptation by the JCE framework.

4.10 edu.rit.m2mp.security.signatures.hors.spec

HORSPrivateKeySpec, and HORSPublicKeySpec Define the basic properties and functionality of public and private keys of the HORS algorithm.

HORSParameterSpec Define the basic properties of keying parameters in the HORS algorithm.

4.11 `edu.rit.m2mp.security.utils`

`Arrays` Support for building larger byte arrays from multiple smaller sequences.

`Numbers` Functionality for reducing and reconstructing multi-byte primitive values into byte arrays of proper length and visa versa.

`ThreadPool` Tunable collection of ready threads. This allows immediate response to new tasks with thread startup overhead penalties only incurred once. This is a stable collection of threads; if a task being run by a thread dies unexpectedly, this pool will initialize a new thread to replace it immediately.

Chapter 5

Operational Overview

5.1 Node Initialization

Each node is uniquely identified by a Java integer value. This value is chosen at random at startup. We assume the likelyhood of two nodes selecting the same identifier is small enough to be ignored, however this could be combatted by selecting a new unique identifier after a timeout period while probing for a group. Following this selection, a set of algorithm parameters to be used in key agreement and signature keypair generation is generated. It is important to note that no keypairs are generated at this point; this is done only after contact with another entity, either a lonely node or group, is established.

The various constructs, such as the key agreement infrastructure and message handling systems are then initialized. Once ready, the Discovery thread is initiated, and the node begins searching for a group.

The Discovery phase, as mentioned above in the discussion of the handler, is simply a repeated broadcast `JoinRequest` message consisting of the current node's ID value and the parameters they have generated. This message is rebroadcast on a periodic interval until either a response is received or the desire to not join any group is registered with the Channel.

When a second member receives this message, one of two protocols follow. If the responding party, for discussion named β, is also alone, the parameters used by the new group are those parameters generated by the node with the larger identifiying number. This is outlined in the *Genesis* section below. If, on the other hand, the responding party is part of a group, β's parameters are used. This is described further in the *Member Operations with Existing Group*s section below.

5.2 Genesis

Genesis occurs at the initial formation of a group; where two or more nodes converge and form common group structures. For discussion, assume two nodes, α with ID value n and β with ID value m where $m > n$, wish to form a group together.

One of three possible scenarios follows:

1. Each will receive the other's request.
2. Node α will receive β's request, but β will not receive α's request (at nearly the same time).
3. Node β will receive α's request, but α will not receive β's request (at nearly the same time).

In the case of the first scenario, both α and β will be able to decide proper group responsibilities. As β's ID is larger, the parameters generated and presented by β will be used by the new group.

The second situation will require an additional membership probe from α (assumably after a timeout has lapsed) to continue the joining process.

In the third scenario, since β's ID is larger, β is able to reply immediately with a `JoinGrant` message to α.

A unicast message is then sent from β to α with confirmation of the parameters by `JoinGrant` message. Since these parameters are likely different from those that were locally generated, keys are generated by both parties with these parameters. β can simply save the local keypair, but α must deliver their new public key to β through a unicast `NewKey` message.

Upon receipt, β then can update the keytree and compute the new group key. The entire tree is delivered from β to α — a new group has been formed.

An important theme through this is a pair-wise communication. This is achieved through synchronization and timeout monitoring to be sure only two parties are forming a group at any given time. It is entirely possible that for an even n nodes, $n/2$ groups are formed. While these groups could begin to merge together with the goal of forming a larger group, this was left for future efforts.

In the event of an odd number, $n - 1$, nodes forming a group, up to $\lfloor (n-1)/2 \rfloor$ small groups will form, leaving one node temporarily out. Once a group has been formed, this last node will join to a group after a response to their `JoinRequest` has been received. Again, these sub-groups could homogonize and form a larger group via group merging.

See Appendix B.1 for an example of three-party Genesis.

5.3 Inner Workings Of Key Agreement

Tree-Based Agreement

In tree-based agreement designs, a logical tree maintains membership and keying information. This implementation employs an array-based storage system, and therefore must also maintain proper relationships among indices (left child of index n is at $2n + 1$, right child is at index $2n + 2$, and a nodes' parent (if $n \neq 0$) resides at index $\lfloor (n - 1)/2 \rfloor$. As compared with a reference-based implementation, an array-based design was both simpler from the perspectives of design and implementation, and proved computationally more efficient. Recursively defined methods to prune and graft subtrees provide a highly efficient and correct solution.

It is important to note that this binary tree must grow at the root when a pre-join tree is complete and full. Growing downward presents an issue of non-paired children, and is therefore unable to compute an intermediate key. Growing at the root inserts a new root and new right subtree, preserving a pairwise key association.

Any specific intermediate private key within the tree is easily computed with specific knowledge of the children at that tree position. The public key from one child and the private key from the other, and through the workings of a standard Diffie-Hellman key agreement round, will generate a shared key, recoverable by both parties.

This implementation of the tree provides two methods used specifically in group key generation – `getRootPath` and `getRootCoPath`. These concepts are described in [24]. The `getRootPath` method traverses the tree from a specified node to the tree root, returning the nodes on the most direct root-ward path. The `getRootCoPath` method also traverses the tree starting from a specific node. Instead of generating a list of linearly traversed nodes, a list of ancestral neighbors is formed and returned. From these two lists, the parental private key is generated.

Below is the pseudocode for the tree-based key agreement mechanism as provided in this work.

```
Node[] path = getRootPath()
Node[] coPath = getRootCoPath()

for i = path.length-1 to 1 {
  path[i-1].keyPair = doAgreement(path[i].getPrivateKey(),
                                  coPath[i].getPublicKey())
}
PrivateKey groupKey = path[0].getPrivateKey()
```

It is critical to remember that a public key is a function of a private key value. To this end, once a private intermediate key has been generated, an associated public key

must be built that is mathematically related to this private value. For both key styles investigated in this work, this meant reaching into other source code. The code from the EC provider was easily ported into this work, but the JCE DSA/RSA key functions required a great deal more effort.

5.4 Member Operations With Existing Group

Post-formation group operations are somewhat more simplified than similar operations done during genesis. Several group-level opeations are defined for this environment – member join, member leave, group merge and group partition. Only the two single member events have been implemented in this work.

When a node announces their desire to join a group, each node tentatively adds the joining member to their group. Only one member, however, will respond and initiate a conversation with the joining member. Key algorithm parameters are sent (by `JoinGrant` message), a public key is received (via `NewKey` message), and the key tree is re-calculated and redistributed (by an `AgreeSet` message) to the group by this sponsor. This recalculation process includes the sponsor generating a new random keypair to use in the generation of the new tree. Figure 5.1 shows this event graphically.

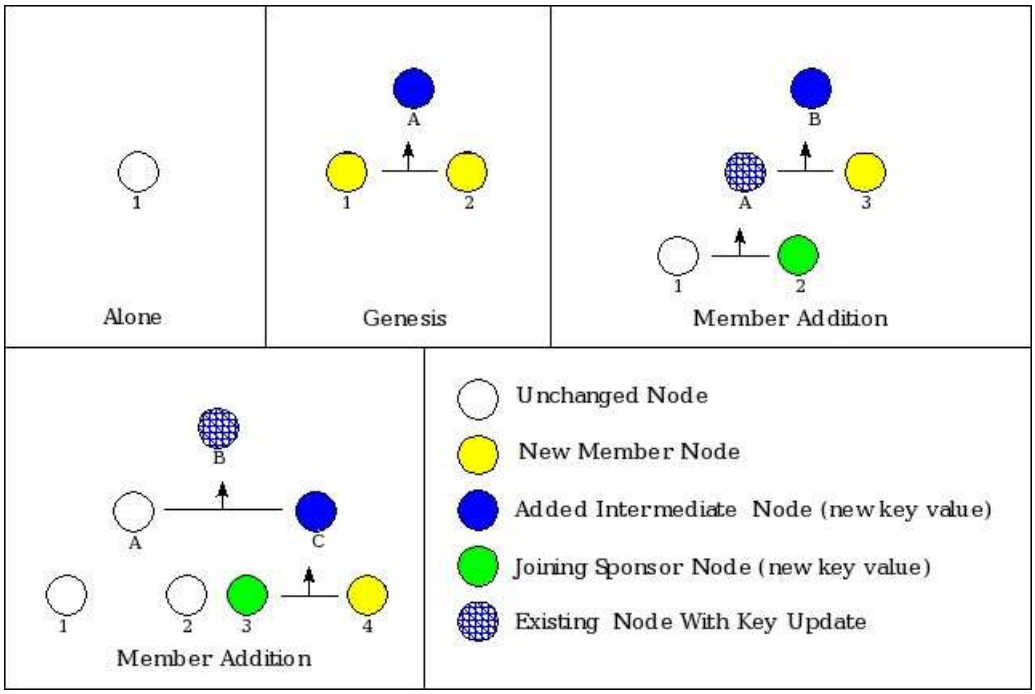

Figure 5.1: Member Join event

Conversely, when a member announces their intentions to leave the group (via a `LeaveRequest` message), the sponsor is again identified and charged with the task of

recalculating and redistributing the key tree. Like member join, the tree recalculation process includes a new sponsor-local keypair to use in tree recalculation. Figure 5.2 shows this event graphically, and Section B.2 provides a transcript of a MemberLeave event.

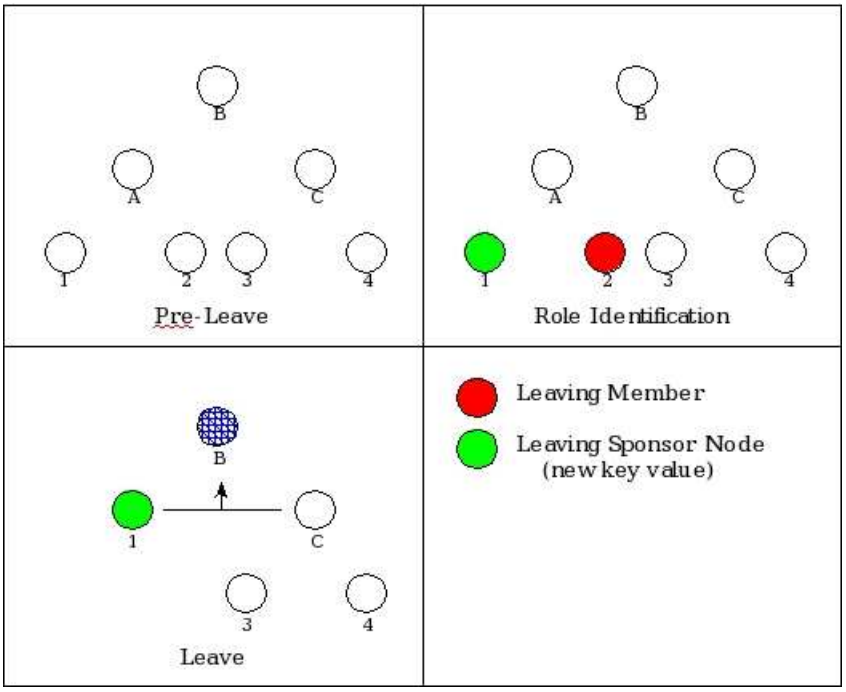

Figure 5.2: Member Leave event

Key update, while not explicitly a member operation, is extremely similar to these two operations. A member requesting (via `KeyUpdate` message) a key update will simply broadcast this message to the group. This member's sponsor will then update the sponsor-local keys, and recompute and redistribute the key tree. While this does not change the entire tree, it updates $\log n$ keys of the tree, including the final group key. Additional work to ensure more widespread key freshness could easily be conducted, such as key age monitoring and periodic key refresh support. Figure 5.3 shows the key update event graphically.

5.5 Signatures

In theory every message received must be authenticated to ensure message integrity and sender validation. In practice, however, this is not possible.

During member join, for example, it is impossible for a non-member to generate a keypair, sign a message, and have the other party validate this signature. The key algorithm is not homogeneous, and the receiving party does not have a viable validation

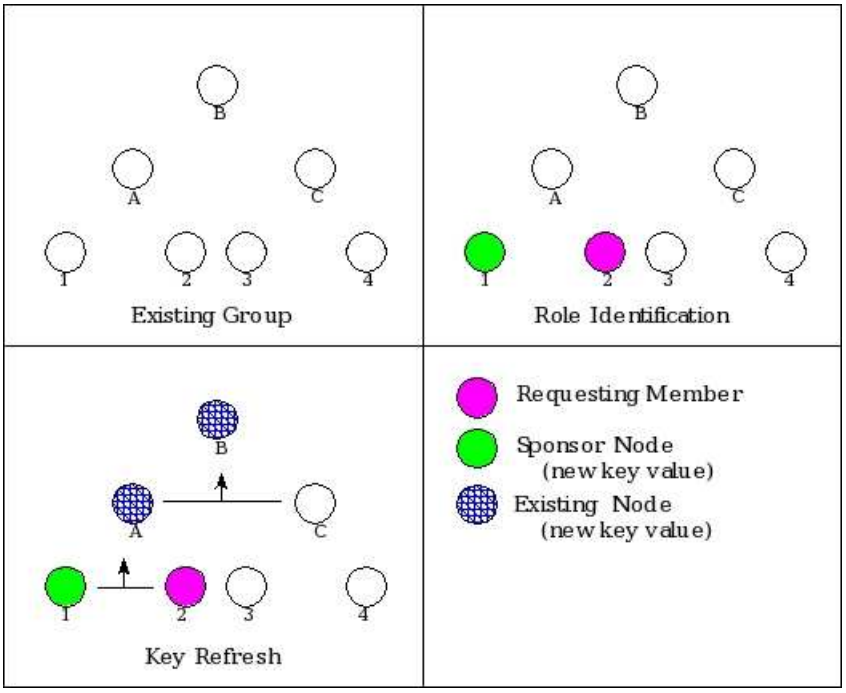

Figure 5.3: Key Refresh event

key. This means that until a member has sent their `NewKey` message and then become a member of the group, these messages should not be signed. If they are signed, the signature should be ignored by the receiving party.

In fact, only messages sent and received as a member of the group should be signed and verified. This enforces message integrity of all messages that could affect the stability and functioning of the group.

Chapter 6

Cryptographic Evaluation

This chapter will address the computational and mathematical foundations of the two key generation and digital signature techniques utilized in this resarch.

6.1 RSA/DSA Algorithmics

RSA - Rivest, Shamir, Alderman Authentication Algorithm

RSA was developed at MIT and released to the public in 1977. With the declassification of government documents in 1977, it was discoverd that British researcher James Ellis developed RSA-like algorithms in 1972. Due to the secrecy of the algorithms at the time, Ellis had no copyright privileges.

RSA keys are generated as follows:

1. With a goal of computing an n-bit key, define large primes p and q where p and q are of bit-length l where $(512 \leq l \leq 1024)$, and $l \approx n/2$.

   The probability a number p_t where $p_t < n$ is prime is roughly $6/n$, and therefore a number will be found in roughly $n/6$ trials. (The number of primes less than n is roughly $n/\ln n$, such that the probability of a number near n being prime is approximately $1/\ln n$. Therefore an $n/2$-bit number has a probability of roughly $1/\ln 2^{n/2}$. Removing even numbers from this field results in a probability of a given number close to $n/2$ being prime being approximately $6/n$).
2. Compute n as $n = pq$.
3. Find e such that e is less than n and relatively prime to $(p-1)(q-1)$.
4. Select another value d such that $(ed-1)$ is divisible by $(p-1)(q-1)$.

The public key value is the value-pair (n, e), and the private key is the value-pair (n, d).

DSA - Digital Signature Algorithm

DSA was designed by the United States federal government to be a very fast and effective signature scheme only. It was later discovered to be a rather effective encryption scheme (a point designers tried to avoid), but proved to be painfully slow. Released to the public in 1991, DSA was not welcomed warmly by a security community that had large financial and technological investments in RSA.

DSA keys are generated as follows:

1. Generate one 160-bit and one 1024-bit prime number as q and p respectivly, ensuring q divides $p-1$.
2. Compute $g = h^{(p-1)/q} \bmod p$ for an h in the cyclic group defined by p.
3. The private key is x and the public key $y = g^x \bmod p$ for a random x in $1 \leq x \leq q-1$.

6.1.1 The Strength of RSA/DSA Schemes

RSA and DSA techniques are based on the difficulty of the Discrete Logarithm Problem (DLP). That is, it is computationally difficult to compute the private value d from the public pair (n, e). This would require factoring the value of $e \bmod n$ into p and q, thereby providing the path to recovering the private key value d. It is this factoring a large value into the product of two primes that is the DLP.

The strength of these schemes rests solely on the size of the public key. As [5] reports, factoring the public key can take only a few hours on a standard PC for key sizes less than 256 bits, which serves as direct motivation for p to be at least 1024 bits.

As DSA parameters and keys are defined over a cyclic sub-group of the integers, DSA is theoretically vulnerable to two discrete logarithm attacks. The first is on the number field itself as defined by the large prime p. Factoring this number will allow recomputation of the private key value. The general number field seive provides the best performace for factoring a large prime-product, running in super-polynomial, sub-exponential time [4]. The second attack uses Pollards-ρ algorithm to compute the value of x as used in the computation of the keypair. Pollards-ρ algorithm for factoring large prime p works as follows [6] (See [39] and [7] for more):

1. Set $a = 2$ and $b = 2$.
2. While not solved and not in error condition, compute $a = a^2 + 1 \bmod p$, $b = b^2 + 1 \bmod p$, and $d = gcd(|a-b|, n)$. If a solution $1 < d < n$ is found, p has been factored into d and p/d. An error surfaces when $d = n$. Additional discussion about this algorithm and suggestions for handling this error condition is available in [6].

This algorithm runs in $O(\sqrt{(\pi n)/2})$ time, and recent work to parallelize the algorithm has improved it's performance to $O((\sqrt{(\pi n)/2})/m)$ for m processors.

6.1.2 Digital Signatures

Given a message m generating a digital signature of m using an RSA/DSA-style private key n, d is computed as follows. A secure hashing algorithm, such as SHA, is applied to the message. This generates a constant-length, collision-resistant bit string representative of m. This hash is then encrypted with the private key, which produces the final message signature. This signature is included in the message as an appendix.

Verification of this signature is the reverse; upon receipt, the signature is decrypted with the public key share to reveal the hash bit string. The original message, minus the included signature, is then processed with the same hashing algorithm. The two hashes are compared, and if equal, verification has passed [5].

6.2 Elliptic Curve Algorithmics

This technique achieves security by being rooted in the Elliptic Curve Discrete Logarithm Problem (ECDLP). The ECDLP is a subproblem of the DLP, and has been proven more difficult to solve, thereby offering higher levels of security than the DLP.

Elliptic curves are mathematically defined as a set of points over a number field (typically the complex, rational, or real numbers) satisfying a cubic curve of genus 1 (one bisecting cut through the curve removes the notions of inside-the-curve and outside-the furve). Elliptic curve equations do not define elliptical paths – they are named for their relation to elliptic integrals used in the computation of elliptical arc lengths, and are closely related to the torus. The complex, rational, and real numbers are, by definition, infinite number fields, whereas the integer family describes an infinite ring (recall fields are a subset of numerical rings, adding the requirement that the non-zero elements of a field form an abelian group under multiplication). Additionally, these number fields (and therefore rings) have the property of either being of characteristic 0 (zero) or of prime characteristic [40]. While other number fields do not share this property, elliptic curves for use in cryptography, regardless of over which field they are defined, must be of characteristic zero or prime. (Fields of characteristic not equal to zero are fields generated by a polynomial or other such functions. All ordered fields are of characteristic zero.) This ensures an abelian group over addition and multiplication.

Curves are typically defined over a cyclic group of prime order or prime-power-of-two order. These fields are represented as E_{F_p} for prime-order fields, and E_{F2^p} for powered-order fields. These fields are each cyclic as they are abelian over addition and multi-

plication (for any two elements A and B in field F, $A \diamond B = B \diamond A$ where $\diamond$ is the binary operation of addition or multiplication), and therefore possess a group generator G from which all elements of the field F may be generated.

Generally elliptic curves (assume characteristic zero), may be written in general form as $y^2 = x^3 - ax - b$. Curves used in cryptography include a point *at infinity*. With two points on the curve, P_{x_P,y_P} and Q_{x_Q,y_Q}, we can then define a third based on the following rules, where the sum of the three points is the point of infinty, denoted $\bigcirc$.

1. If $P \neq Q$, R' is the point defined by drawing a line intersecting the curve at points P, Q, and R'.
2. If the intersecting line is tangential to the curve at either point P or Q, this point in included twice.
3. If the connecting line is parallel to the y-axis, the third point is the point of infinity.

Given an elliptic curve E and two points on this curve, P_{x_P,y_P} and Q_{x_Q,y_Q}, the sum of these points is $P_{x_P,y_P} + Q_{x_Q,y_Q} = R'_{x_R,y_R}$. The point R' is determined by 'drawing' a line segment between P_{x_P,y_P} and Q_{x_Q,y_Q}. This line segment will intersect the curve elsewhere at a third point, R'_{x_R,y_R}. Reflecting the y-component of the coordinates of the point R' defines the sum-point R_{x_R,y_R}.

This translates computationally to (see [41], [2]):

1. Assume two points P_{x_P,y_P} and Q_{x_Q,y_Q} reside along a curve, and we want $P_{x_P,y_P} + Q_{x_Q,y_Q} = R_{x_R,y_R}$.
2. Calculate the slope of the connecting line as $s = (P_y - Q_y)/(P_x - Q_x)$.
3. If $P_x \neq Q_x$, calculate the final coordinates as
   $R_x = s^2 - P_x - Q_x$ and
   $R_y = -P_y + s(P_x - R_x)$.
4. If, on the other hand $P_x == Q_x$, there are two paths:
   (a) If $P_y = -Q_y$ (a P-Q connecting line would be parallel to the y-axis), then $s = (3P_x^2 - a)/(2P_y)$ (recall the value of a is from the curve generation sequence)
   $R_x = s^2 - P_x - Q_x$ and
   $R_y = -P_y + s(R_x - P_x)$
   (b) Otherwise, as mentioned above in the discussion of the point of infinity, when the y-coordinates of both points match (this can only happen at the point of infinity or when computing a scalar multiple of a point), $R = 2P = 0$.

Subtraction of P_{x_P,y_P} from R_{x_R,y_R} (where $P_{x_P,y_P} + Q_{x_Q,y_Q} = R_{x_R,y_R}$) is done similarly. Reflecting the y-component of R point to describe the point R', connecting R' and Q with a straight line will describe the point P – the difference between points Q and R.

Scalar multiplication, then, is a series of doublings and additions. Again, with elliptic curve E and a point P_{x_P,y_P}, $2P_{x_P,y_P}$ is simply $P_{x_P,y_P} + P_{x_P,y_P}$. We find the point of intersection, Q_{x_Q,y_Q}, the tangent line at P_{x_P,y_P} makes with E, and negate the y-coordinate, giving us our final point R_{x_R,y_R}. It should be noted that $3P_{x_P,y_P}$ is simply $2P_{x_P,y_P} + P_{x_P,y_P}$.

The example is given in [18] of computing $11P$ for a point P on any curve as $11P = (2 * ((2 * (2 * P)) + P)) + P$ (Extra parenthasis are present to force proper mathematical ordering). This approach works to calculate $11P/2$ as quickly as possible with the fewest calculations. The value of $11P$ could also be calculated as $11P = (2*(2*(2*P)))+(2*P)+P$, but this is clearly not as clean, nor as straighforward to implement, especially with large coefficients.

This information, along with a great deal of other pertinant information, is explained in greater depth by [8], [41], [40], [2] and [3].

6.2.1 Prime-Order Elliptic Curves

Curves defined over the prime order group are defined and generated as follows:

1. Random prime number p greater than 3. The field is then defined over the integers $0, 1, 2, \ldots, p-1$.

   Values should be large enough (at least 224 bits) to not fall prey to the Pohlig-Hellman solution [17] or the birthday attack [1] for solving some discrete logarithm problems.
2. Addition of field elements is done as integer addition $\mathrm{mod} p$.
3. Multiplication of field elements is done as integer multiplication $\mathrm{mod} p$.
4. Two field elements, a and b (positive or negative), must satisfy the equation $4a^3 + 27b^2 \not\equiv 0(\mathrm{mod} p)$.
5. A generator point G, defined by the coordinates (x_G, y_G) that lies on the curve E.
6. The number of bits, n, of the point G.
7. The curve E is then comprised of all points that satisfy the equation $y^2 = x^3 + ax + b$, with the exception of the point of infinity $\bigcirc$, which defines the additive identity of the curve group.

   The number of points along the curve is the *order* of the curve, represented as $\#E(F_p)$. The Hasse Theorem defines $p + 1 - 2\sqrt{p} \leq \#E(F_p \leq p + 1 + 2\sqrt{p}$ [7].

6.2.2 Prime-Power Elliptic Curves

Curves over the prime-power-of-two group are defined as follows:

1. Prime number p greater than 3. The field is then defined over the integers of p-bits in length.
2. Addition of field elements is done as integer addition $\mathrm{mod} p$.
3. Multiplication of field elements is done as integer multiplication $\mathrm{mod} p$.
4. Two field elements, a and b, positive or negative.
5. The curve E is then comprised of all points, with the exception of the point of infinity, which satisfy the equation $y^2 + xy = x^3 + ax^2 + b$.
6. A generator point G, defined by the coordinates (x_G, y_G) that lies on the curve E.
7. The number of bits, n, of the point G.

6.2.3 Common Ground

Keys are then generated as follows:

1. Choose a random value d over the interval $(1 \ldots, n-1)$.
2. Compute the point Q comprised of the on-curve coordinates (x_Q, y_Q) as $Q = dG$.

The public key value is Q, and the private key value is d.

6.2.4 The Strength of Elliptic Curve

The Elliptic Curve Discrete Logarithm problem is simply to find the integer d that represents both the private key value and the factor used in the public key calculation, given only the final public key value. That is, given $Q = dG \bmod p$ and G, find the value of d.

Several different types of valid curves exist that have been removed from consideration in generating curves for cryptographic use. These families are so removed as they have common attributes that are easily exploited, thus violating the security guarantees of the curve.

Whereas RSA/DSA style keys rely on modular integer exponentiation, elliptic curves rely on scalar point multiplication. Any even multiple of a point is a sequence of point doubling, and an odd multiple is a sequence of point doublings and point additions.

On first inspection, the Discrete Logarithm Problem and the Elliptic Curve Discrete Logarithm Problem seem nearly identical in representation and definition. However, as [38] explains, through the index calculus attack (the most efficient/successful factoring

technique for large primes currently known), a number may be factored by dividing off prime factors incrementally. Done in a loop until completion, smaller values usually result from this division. Elliptic curves, on the other hand, do not have this decreasing value guarantee. Subtracting a point with small coordinates with a second small-coordinate point, may result in a large-value point. It is for this reason that the ECDLP is considered a subset of the DLP, and computationally harder.

As shown by Figure 6.1, a large division in key sizes for equivalent security is apparent. This shows that elliptic curve-based keys are physically and logically smaller for equivalent key strength. Table 6.1 shows a tabular form of the same data [18]. These numbers represent approximate equivalencies in resitstance to brute-force attacks in efforts to factor the private key value. This is further explained and discussed in [26].

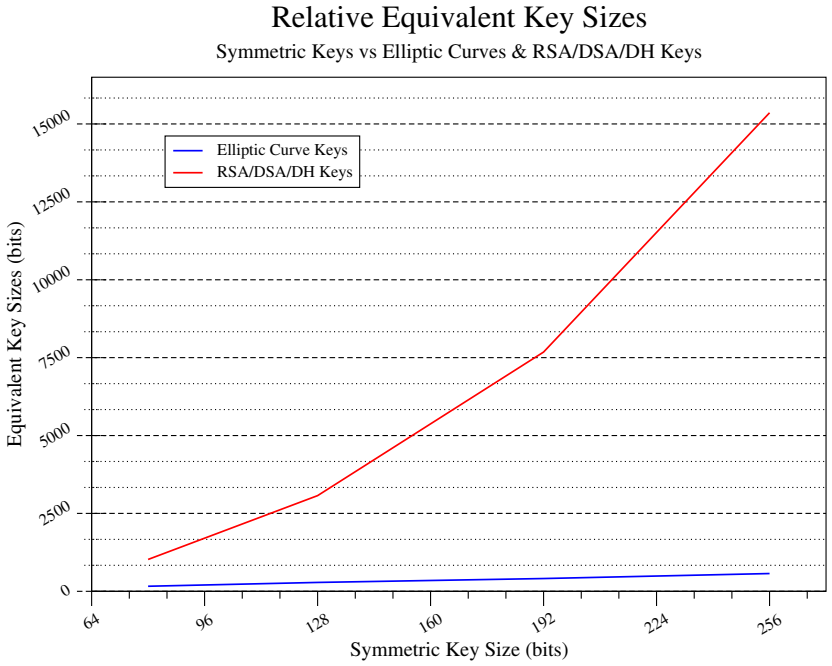

Figure 6.1: Equivalent Key Sizes

Symmetric Key Size	Elliptic Curve Key Size	RSA/DSA Key Size
80	163	1024
128	283	3072
192	409	7680
256	571	15360

Table 6.1: Equivalent key strength comparison

6.3 Diffie-Hellman Key Agreement Techniques

The Diffie-Hellman key agreement algorithm defines a way for two parties to generate a shared key, independently, without exchanging secret information. This allows these same two parties to generate a key in the open, and a third party, who has gathered all exchanges, is unable to generate the same key.

This technique was introducted in 1976 by Whitfield Diffie and Martin Hellman, as published in *"New Horizons in Cryptography"* [14], and has served as the cornerstone technique for two-party key agreement. It has since been generalized with several different variations, some providing authentication and better utilization of certain resources (see [9]) that work better in multi-party settings.

It is important to note that in no way does the standard Diffie-Hellman key agreement algorithm provide endpoint authentication. That is to say that if Alice and Bob are exchanging a key, Eve can situate herself between Alice and Bob and play the role of Bob to Alice, and Alice to Bob. This would allow Eve to view or modify all data sent between Alice and Bob, and nobody would be any the wiser. This is a man-in-the-middle attack. Some of the variations mentioned in [9] address these issues, and combining key agreement with certificate validation is yet another common alternative. While certificates are likely exterior to this kind of computing environment, other key agreement schemes could be explored.

RSA/DSA Key Agreement

Given two keypairs e_α, d_α and e_β, d_β, nodes α and β are each able to individually compute the same common shared key, S. Node α computes $S_\alpha = d_\alpha^{e_\beta} \bmod n$, and β computes $S_\beta = d_\beta^{e_\alpha} \bmod n$. In both computations, S_α and S_β are equivalent. It is clear that a third party, whom has gathered e_α and e_β would compute $S_\gamma = e_\beta^{d_\alpha} \bmod n$, which is clearly not the same.

Elliptic Curve Key Agreement

Given two keypairs Q_α, d_α and Q_β, d_β, nodes α and β are each able to compute ethe same common shared key, S. Node α computes $S_\alpha = d_\alpha Q_\beta$ and β computes $S_\beta = d_\beta Q_\alpha$. In both computations, S_α and S_β are equivalent. It is clear that a third party, whom has gathered Q_α and Q_β would compute $S_\gamma = Q_\alpha Q_\beta$, which is clearly not the same.

This technique extends to the generalized Diffie-Hellman (GDH) [9] technique applicable to groups through a sequence of two-party agreements.

The math involved in the Diffie-Hellman key agreement algorithm clearly shows the importance of the associative and commutative properties of the integer field for RSA/DSA algorithms, and abelian groups for elliptic curves.

6.3.1 Tree-Based Key Agreement

As mentioned above, it is imperative that the tree maintain a paired association of nodes within the tree. This is a direct result of the pair-based key agreement algorithms, discussed in Section 6.3.

Tree-based agreement is best explained by example. Consider a set of nodes $1, 2, 3, \ldots, n$ wishing to form a group. As per the genesis protocol, suppose 1 and 2 form the initial group. They compute a shared key $S_{1:2}$ as described above, saving the value in an intermediate parent tree node. When 3 joins, 3 computes a common key $S_{(1:2):3}$ in the pairwise fashion. This value is then saved as a new intermediate parental node to both the $1:2$ node and 3 node. This generates a common *secret* key, i.e. private key. To provide a public key value at this node, a random public key (yet algabraically related to the private key) is generated and stored. This intermediate public key is generated from and by identical key material as any non-intermediate (i.e., node) keypair. As such, it is just as secure and provides equivalent security utility as any other key in the framework. With a group size of at least three members, this calculation is done by the tree sponsor (see Section 5.4) and then distributed to all group members. A two-member group does not need this distribution as each member is computing the same final key. As additional members join the group, the tree grows at the root, and fills in a top-to-bottom, left-to-right manner. For example, when N^4 joins the group, the tree is rearranged so that the root node has two intermediate childen. The left child of root is parent to nodes N^1 and N^2, and the right of root child is parent to nodes N^3 and N^4. The value of the final group key with four members is (with $S_{N^{1-2}}$ computed by N^1 and $S_{N^{3-4}}$ computed by N^3):

$$\text{If } S_{1:2} = (d_1 Q_2) \text{ and } S_{3:4} = (d_3 Q_4), \text{ then } S_{1:4} = (d_{S_{1:2}} Q_{S3:4}) = (d_{S_{3:4}} Q_{S1:2})$$

Each node would compute the key as follows:

$$\begin{aligned} 1 \text{ computes } S_{1:4} \text{ as } (d_1 S_2) Q_{S_{3:4}} \\ 2 \text{ computes } S_{1:4} \text{ as } (d_2 S_1) Q_{S_{3:4}} \\ 3 \text{ computes } S_{1:4} \text{ as } (d_3 S_4) Q_{S_{1:2}} \\ 4 \text{ computes } S_{1:4} \text{ as } (d_4 S_3) Q_{S_{1:2}} \end{aligned}$$

By leveraging the mathematical properties of the binary tree, we can efficiently compute a common group key in $\log n$ agreement rounds for an n-member group. There are issues surrounding this technique, discussed briefly in Section 8.2, and further in [19].

Section 5.4 outlines these processes in additional detail.

Part III

Conclusions

Chapter 7

Performance

The goal of this research was to support the claim that elliptic curve keys were more suitable for small devices than RSA/DSA-style keys, but not at the cost of security.

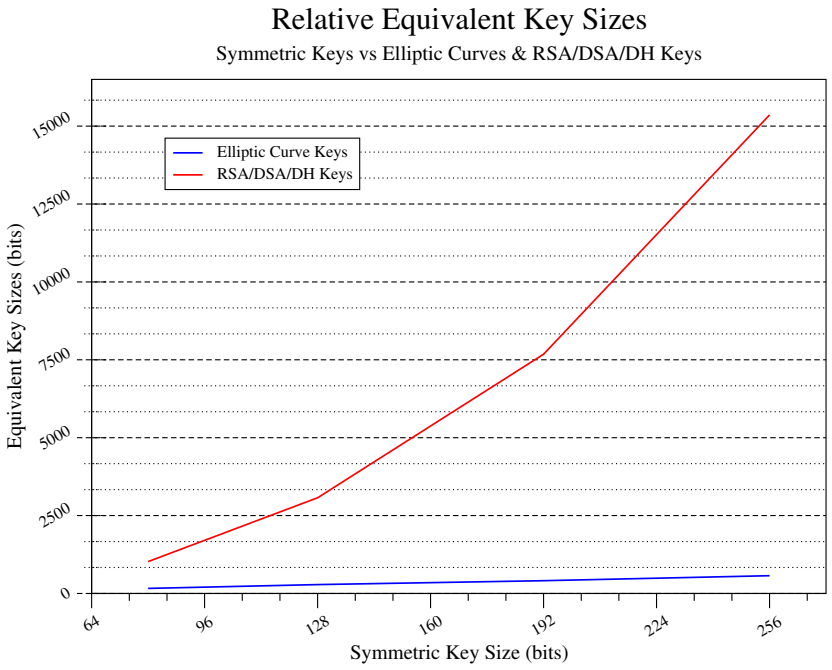

Figure 7.1: Equivalent Key Sizes

Additionally, a key strength sufficient for our needs, yet not suseptible to compromise must be identified. While the framework implemented in the thesis is targeted at small devices, there is no stipulation that larger systems cannot participate in the group. This larger system may have facilities to attack weak cryptography, so our cipher must be mathematically and computationally strong. Figure 7.1 shows the growth rates in security of both RSA/DSA-style keys and EC keys for the most common symmetric key sizes.

In [11], the authors point out that an elliptic curve key is proportional to an RSA/DSA

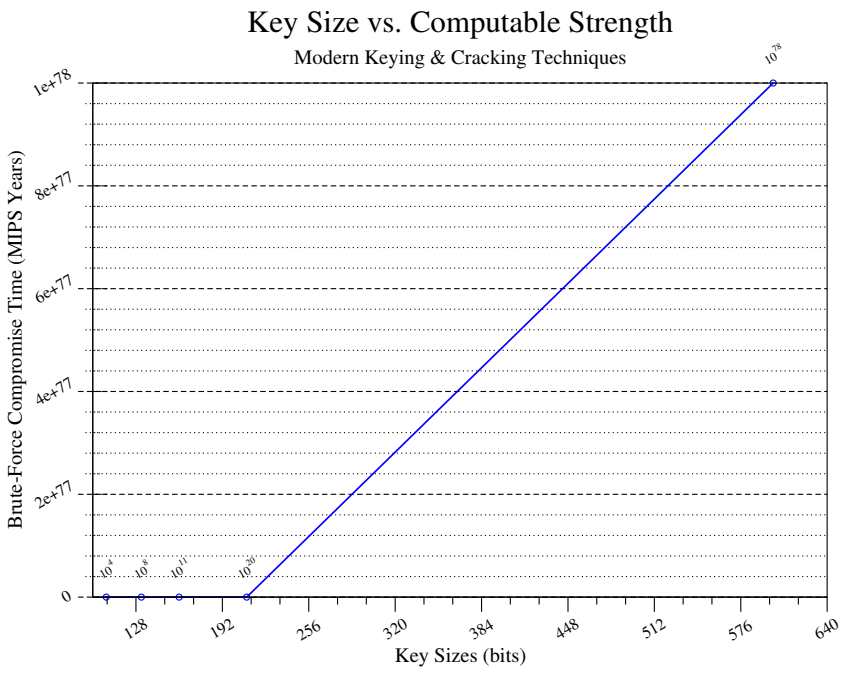

Figure 7.2: Computability comparison and key strength

style key according to the equation:

$$n = \beta N^{1/3}(ln(Nln(2)))^{2/3}$$

where $\beta \approx 4.91$, and N is the RSA/DSA key size. The value of β is also fully explained in [11].

This ratio generalizes the size-strength differences between RSA/DSA- and elliptic curve-based keys to show that curve based systems grow slightly faster than the cube root of DSA/RSA key sizes while maintaining approximately equal strength.

7.1 General Key Generation

The following two graphs, Figures 7.4 and 7.5, plot the maximum, minimum, and average times needed to generate a keypair using the two different techniques (RSA/DSA vs elliptic curve), over varying key sizes. It is clear from the data that generating a JCE key (RSA/DSA style) can take significantly longer than an elliptic curve-based key.

Times are tallied from a series of ten key generations for each key size, with times reflecting JCE provider load and general initialization removed.

Figure 7.3 shows the DER-encoded (serialized) sizes of the various key sizes for both algorithms. Smaller is better, as the lengths represented are per-key sizes as transferred during key serialization operations.

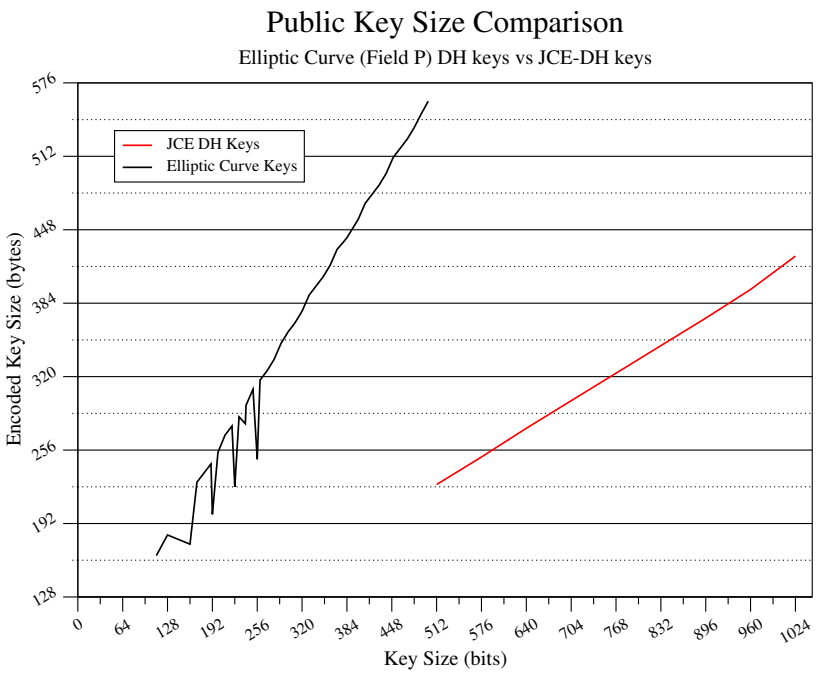

Figure 7.3: DER-Encoded Public Key Size Comparison

7.1.1 EC-Based Keys

Figure 7.4 is shows times needed to generate various F_p-field keys. The tested version of the FlexiECProvider [16] does not support curves over the F_{2^p} field.

7.1.2 JCE Keys

The timing values are shown in Table 7.3; key sizes are in bits, times are measures of seconds. It is clear that JCE keys can take from thirty seconds to seven-plus minutes to generate on a desktop system. Also important to note with Figure 7.5 is that the y-axis is logarithmic. The disparity between times for RSA/DSA and elliptic curve keys is made very evident by the composite graph, Figure 7.6. The sizes for EC-based keys have been adjusted (according to [26]) to reflect similar cryptographic strength as their RSA counterparts (i.e., a 160-bit elliptic-curve key is approximately cryptographically equivalent to a 1024-bit RSA key).

7.2 Digital Signatures

The other significant part of this work is the digital signature utilization investigation. The following graphs plotting signature timing performance was based on the following test routines: at 1024-byte increments, starting at 1024 bytes and progressing to one

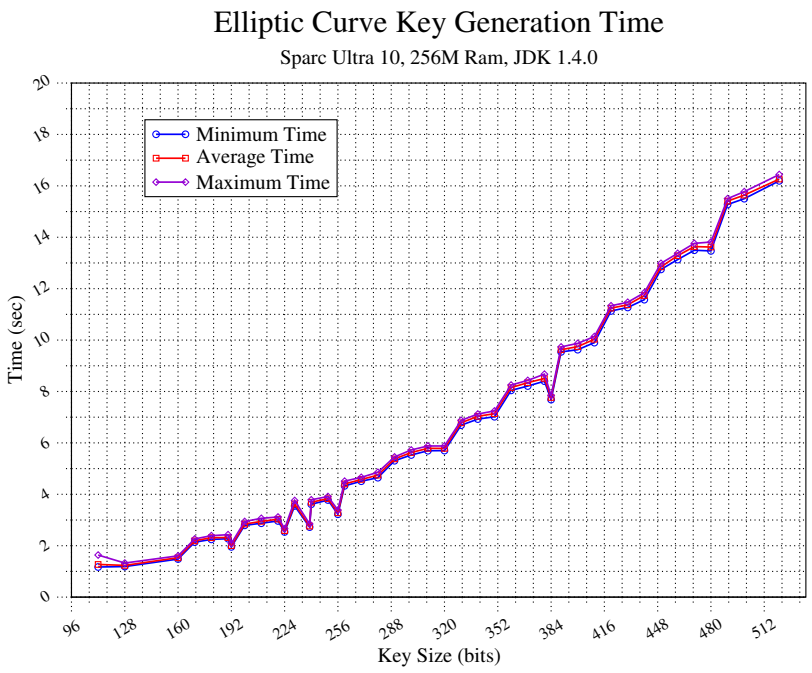

Figure 7.4: EC key generation timings

megabyte, generate random data of proper length. Time the signature update and signing phases for one hundred iterations. All tests use the same key, and timing does not include random data generation. All tests at each data-length use the same random data. The tests were conducted on a Sun Sparc Ultra 10, and an AMD Athlon MP 1800+ to help ensure performance results are platform independent.

Digital signatures first reduce an arbitrary-length data stream to a fixed-length stream using a message digest hashing technique. This is typically done with SHA or the one of the classic MD alogrithms (MD4, MD5). This constant-length hash value is then encrypted with the private signature key. This provides a collision-resistant base to encrypt in constant time, with linear overall complexity (linear in data length for hashing).

SHA operates on blocks of data, padding to a fixed size as needed. The Java implementation of SHA as called from both signature schemes is SHA-1 (see [27] for more about SHA-1).

7.2.1 Architecture 1 - Sparc

Sun Sparc Ultra 10, with 256 megabytes of memory using JDK 1.4.0 running Solaris 9. This was the initial testbed for this examination.

Visible in the following graphs (Figures 7.8 and 7.9), there are operations within SHA that cause severe performance degredation on this platform. These may include data swapping or caching events. Comparativly, in graphs from the second architecture

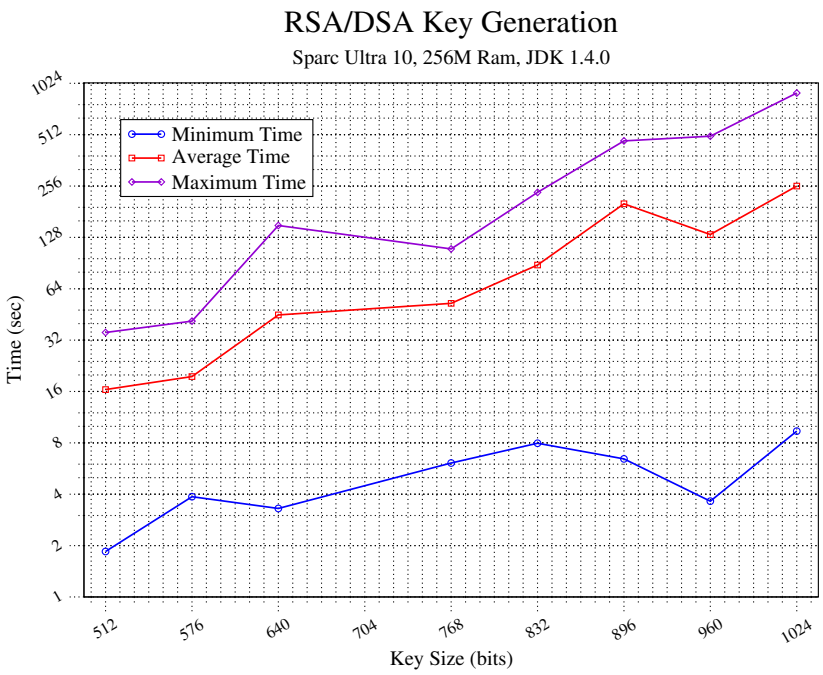

Figure 7.5: JCE key generation timings

(Figures 7.15 and 7.16), these influences are less visible, and the blocking process is clearly visible in the stepping advancement of the datapoints.

Figure 7.7 shows all three (minimum, average, and maximum) times for total signature generation on the Sparc-based system using EC-based signatures.

Figure 7.8 shows the minimum and average times for both the hashing and encryption phases on the Sparc-based system using EC-based signatures.

Figure 7.9 shows the maximum times for both the hashing and encryption phases on the Sparc-based system for EC signatures.

Figure 7.10 shows all three (minimum, average, and maximum) times for total signature generation on the Sparc-based system using RSA-based signatures.

Figure 7.11 shows the minimum and average times for both the hashing and encryption phases on the Sparc-based system using RSA/DSA-style signatures.

Figure 7.12 shows the maximum times for both the hashing and encryption phases on the Sparc-based system using RSA/DSA-style signatures.

Figure 7.13 clearly shows the performance differences (based on average times) of EC- and RSA-based digital signature schemes on the Sparc system.

7.2.2 Architecture 2 - AMD

AMD Athlon MP 1800+, with 512 megabytes of memory using JDK 1.4.2 running the Linux 2.6.5 kernel. With the results from the Sparc-based tests, this examination was

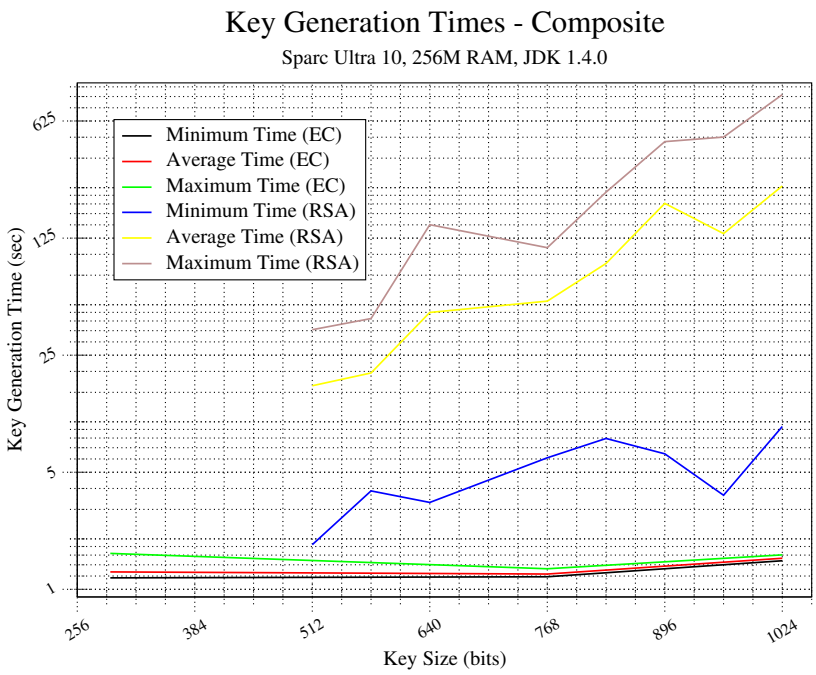

Figure 7.6: Key generation timings

performed to determine the influences of the machine architecture. It is clear from the following graphs there are some aspects of this signature process that are influenced by the system, not just overall procesing speed.

Figure 7.14 shows all three (minimum, average, and maximum) times for total signature generation on the AMD-based system using EC-based signatures.

Figure 7.15 shows the minimum and average times for both the hashing and encryption phases on the AMD-based system using EC-based signatures.

Figure 7.16 shows the maximum times for both the hashing and encryption phases on the AMD-based system for EC signatures.

Figure 7.17 shows all three (minimum, average, and maximum) times for total signature generation on the AMD-based system using RSA-based signatures.

Figure 7.18 shows the minimum and average times for both the hashing and encryption phases on the AMD-based system using RSA/DSA-style signatures.

Figure 7.19 shows the maximum times for both the hashing and encryption phases on the AMD-based system using RSA/DSA-style signatures.

Figure 7.20 clearly shows the performance differences (based on average times) of EC- and RSA-based digital signature schemes on the AMD system.

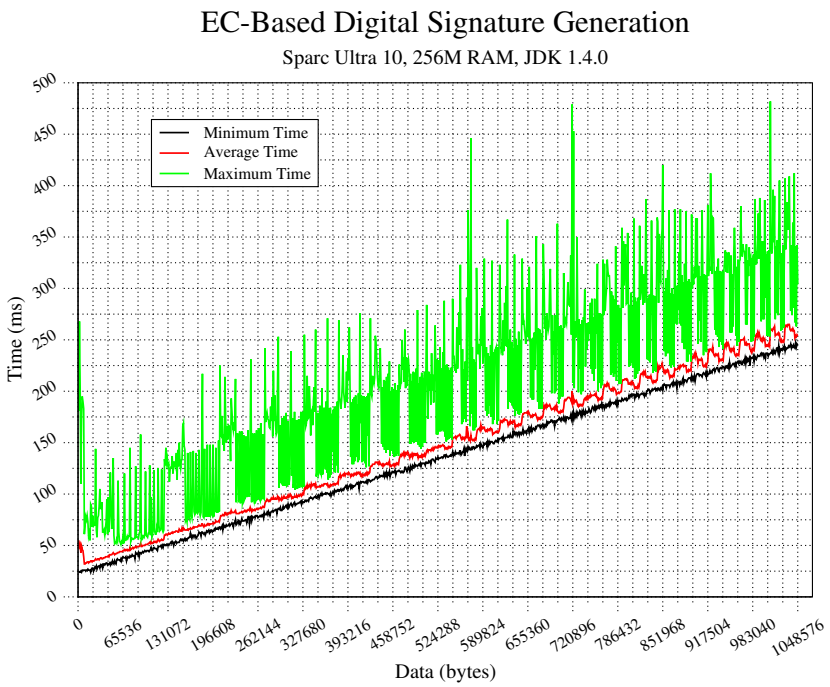

Figure 7.7: EC-Based Signature Timing - Min, Max, Avg (Sparc)

7.3 Conclusions

From this research, it is clear that elliptic curve keys are more suited for a computationally-limited network environment than RSA/DSA style keys. Faster generation times and simpler algorithmics place substantially less strain on small devices, and make group participation possible.

Furthermore, the ability to compute and verify digital signatures with the same key-pair is a feature other key generation styles simply cannot provide due to their mathematical roots. This reduces key distribution and synchronization costs by half within the group, and provides an easy way for any group member to update the group key at any time.

An intelligent key agreement scheme and inner-group maintenance protocol ensures group stability as well as perfect forward and backward secrecy. Using the same keys for both key agreement and digital signatures requires all members possess a current and complete view of the group; this further promotes group-wide stability and security as no one member holds a monopoly on the groups' cryptographic core.

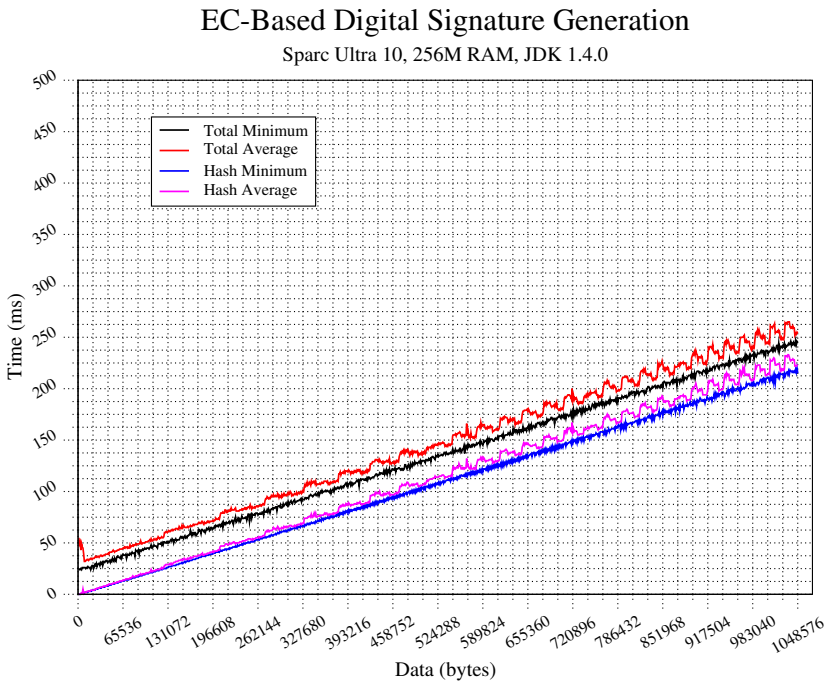

Figure 7.8: EC-Based Signature Timing - Min, Avg (Sparc)

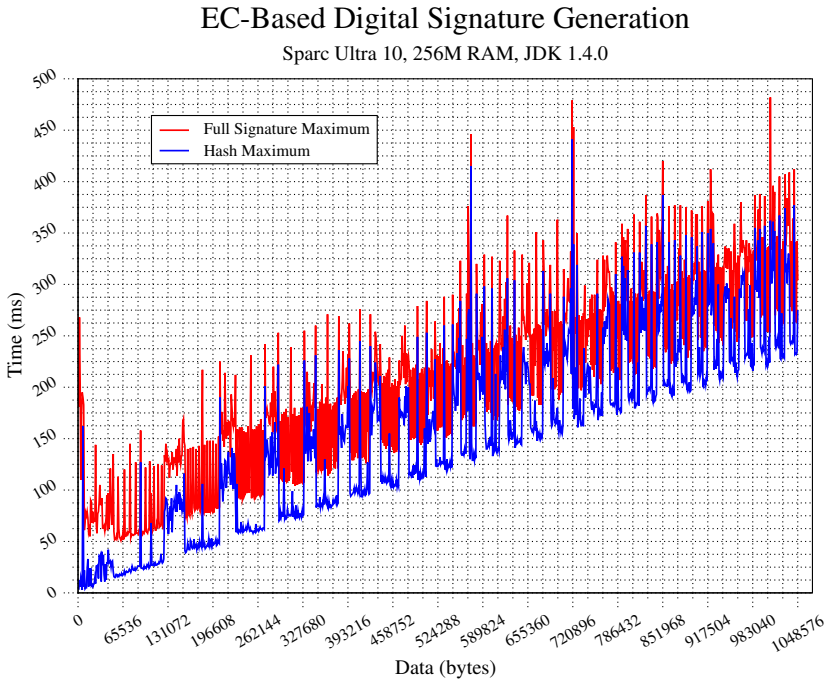

Figure 7.9: EC-Based Signature Timing - Max (Sparc)

Key Size	Miminum Time	Average Time	Maximum Time
112	1170	1270	1638
128	1188	1233	1326
160	1481	1533	1605
170	2136	2190	2264
180	2246	2313	2391
190	2271	2316	2429
192	1953	2003	2088
200	2790	2844	2944
210	2874	2946	3066
220	2963	3031	3116
224	2534	2586	2663
230	3545	3627	3760
239	2722	2764	2847
240	3612	3690	3787
250	3776	3831	3914
256	3222	3284	3387
260	4329	4402	4510
270	4509	4578	4659
280	4646	4747	4856
290	5305	5377	5453
300	5529	5626	5727
310	5694	5788	5880
320	5701	5787	5880
330	6688	6782	6876
340	6926	7035	7124
350	7018	7138	7242
360	8044	8152	8251
370	8212	8343	8431
380	8399	8506	8674
384	7683	7767	7844
390	9541	9626	9733

Table 7.1: Elliptic Curve Key Generation Timings

Key Size	Miminum Time	Average Time	Maximum Time
400	9626	9747	9870
410	9901	10031	10136
420	11131	11243	11335
430	11268	11371	11472
440	11571	11733	11844
450	12750	12848	12985
460	13130	13283	13375
470	13493	13637	13763
480	13469	13620	13815
490	15275	15407	15503
500	15499	15641	15773
521	16193	16263	16439

Table 7.2: Elliptic Curve Key Generation Timings (cont.)

Key Size	Miminum Time	Average Time	Maximum Time
512	1847	16416	35462
576	3868	19571	41344
640	3301	44971	150264
768	6099	52560	109521
832	7955	88176	235043
896	6441	201450	470810
960	3642	133015	501504
1024	9373	255904	898137

Table 7.3: JCE Key Generation Timings

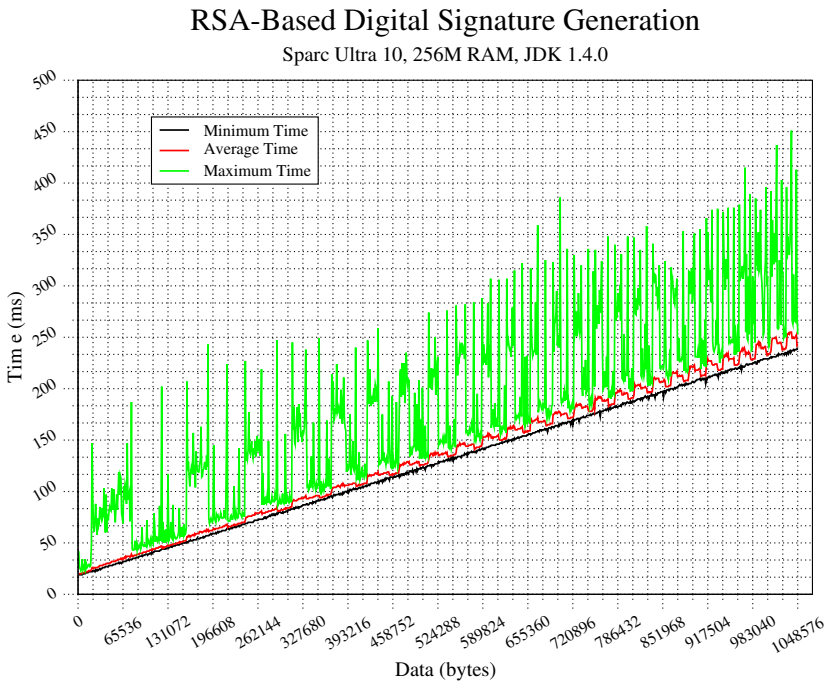

Figure 7.10: RSA-Based Signature Timing - Min, Max, Avg (Sparc)

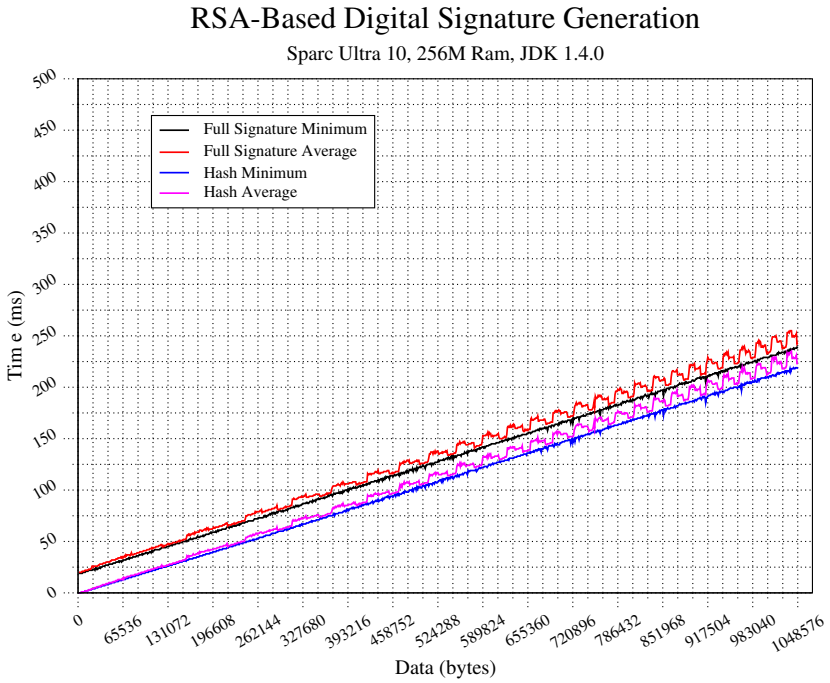

Figure 7.11: RSA-Based Signature Timing - Min, Avg (Sparc)

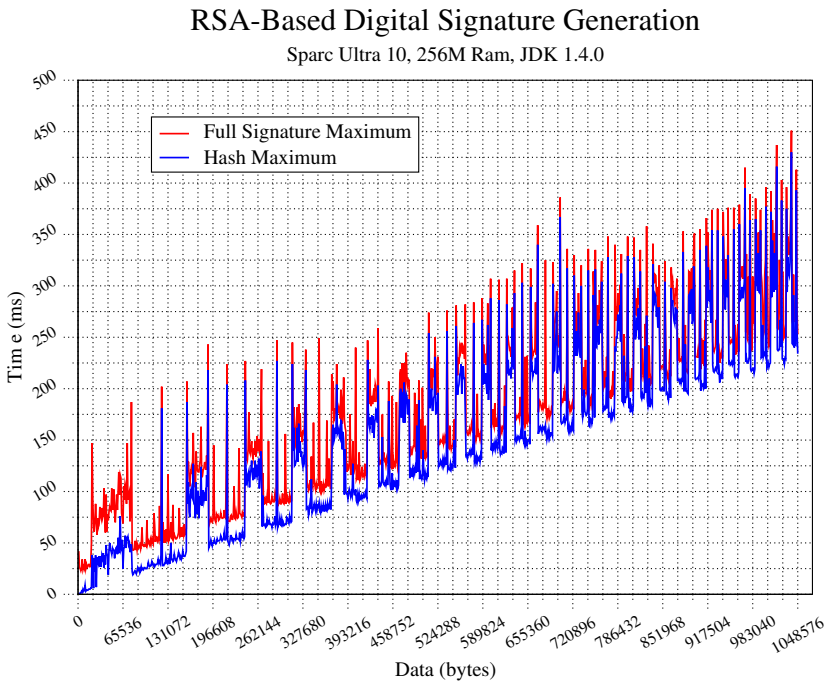

Figure 7.12: RSA-Based Signature Timing - Max (Sparc)

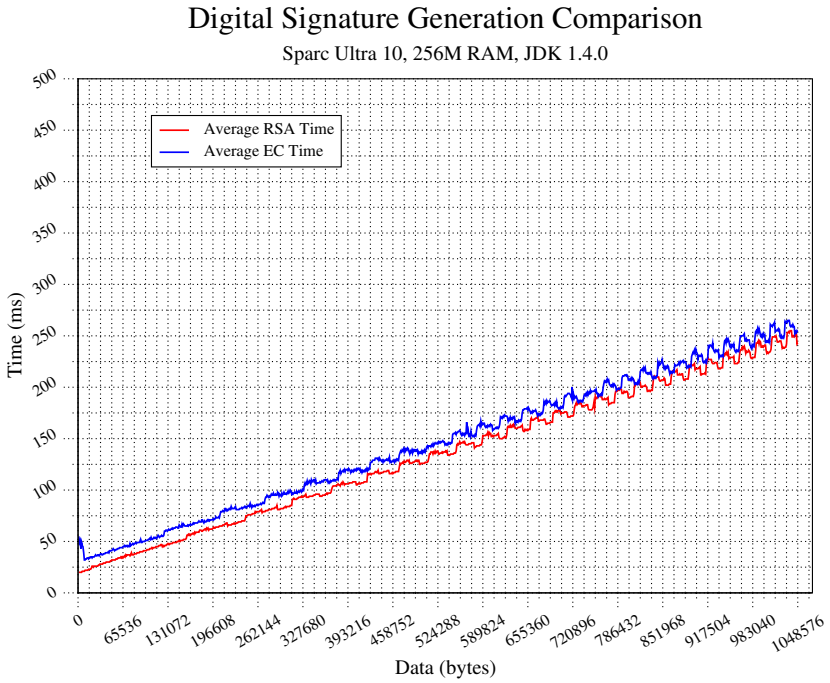

Figure 7.13: Signature Timing Comparison - Avg (Sparc)

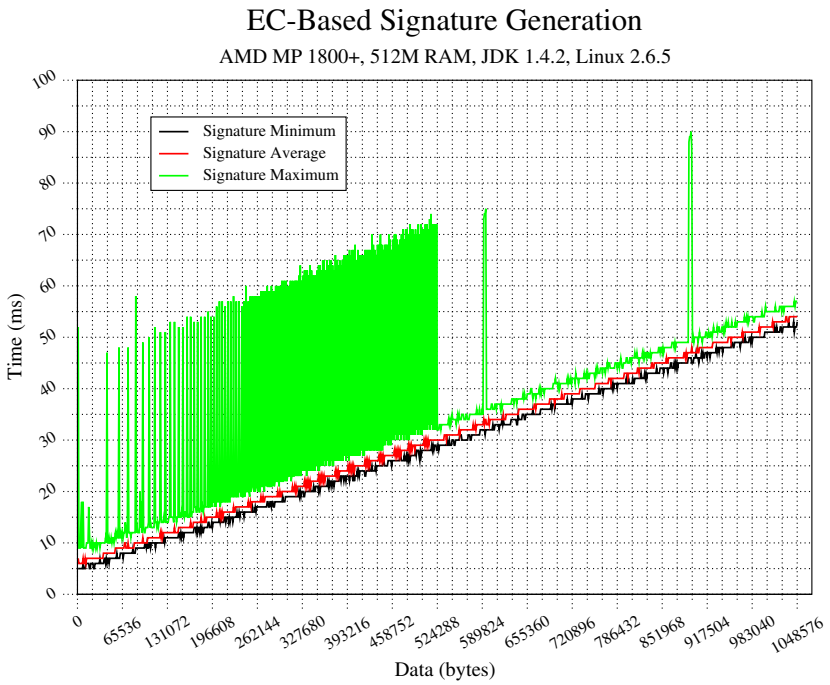

Figure 7.14: EC-Based Signature Timing - Min, Max, Avg (AMD)

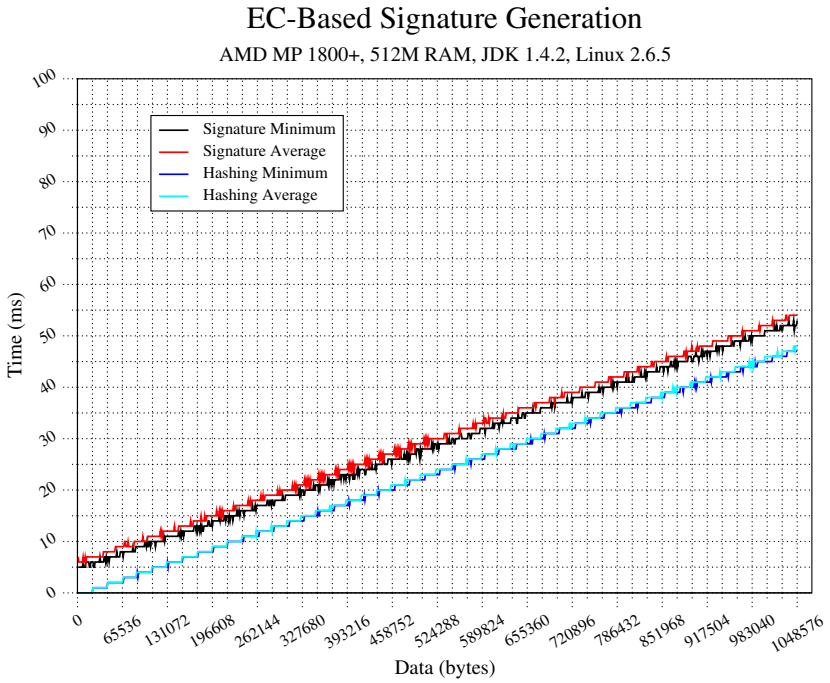

Figure 7.15: EC-Based Signature Timing - Min, Avg (AMD)

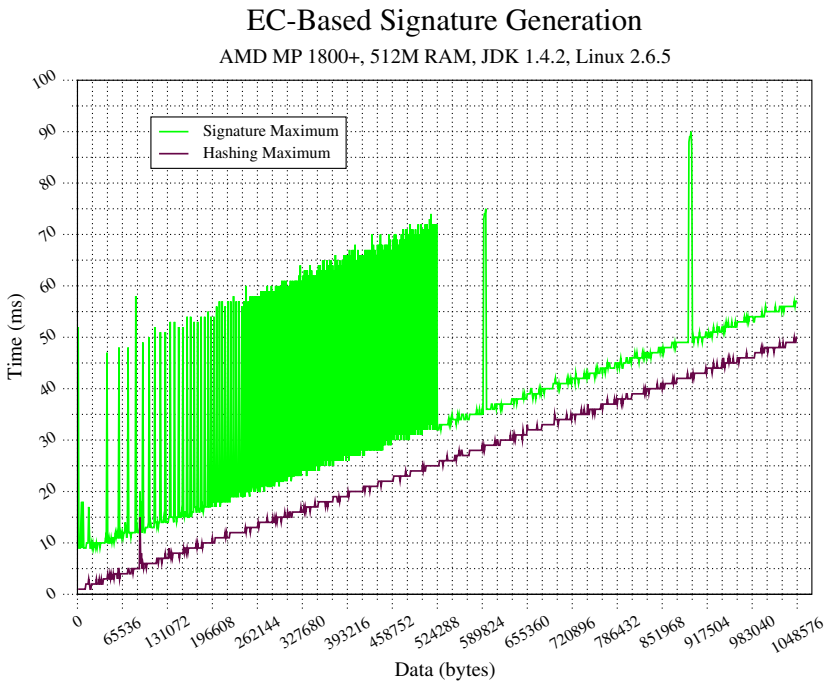

Figure 7.16: EC-Based Signature Timing - Max (AMD)

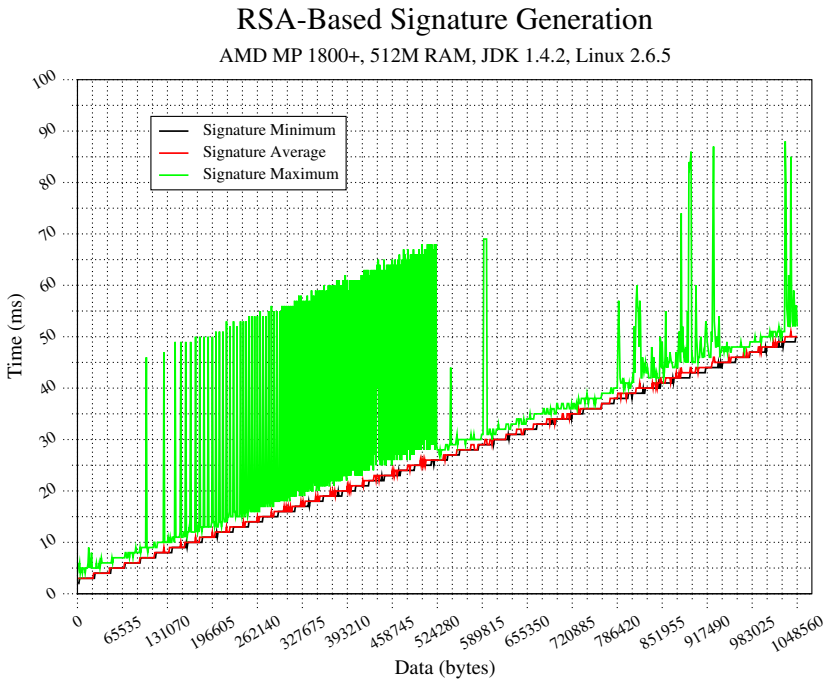

Figure 7.17: RSA-Based Signature Timing - Min, Max, Avg (AMD)

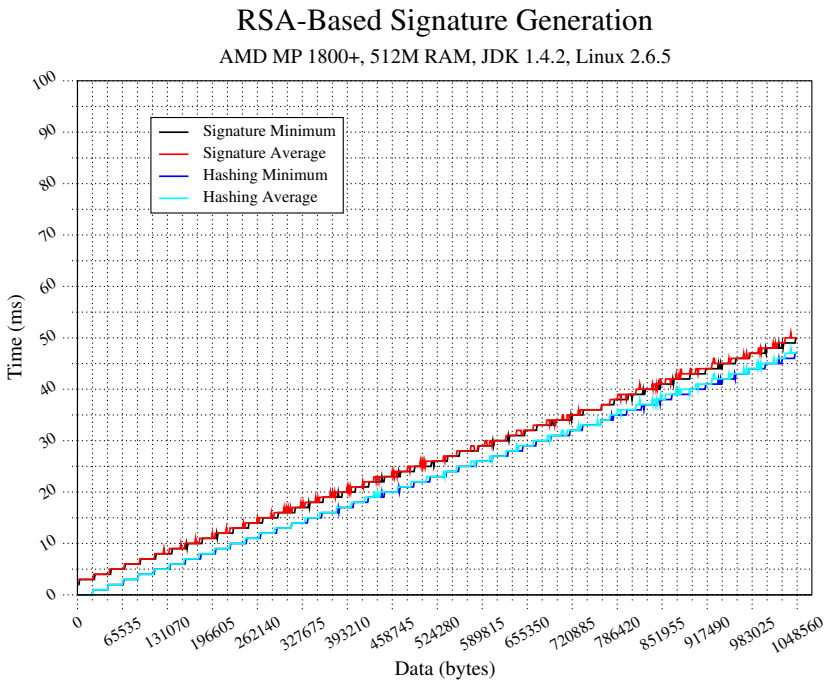

Figure 7.18: RSA-Based Signature Timing - Min, Avg (AMD)

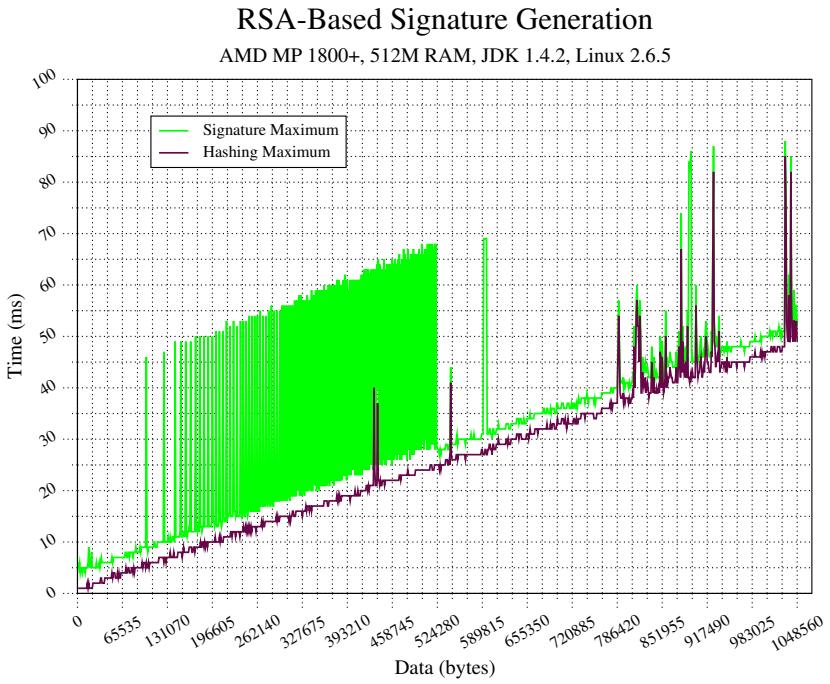

Figure 7.19: RSA-Based Signature Timing - Max (AMD)

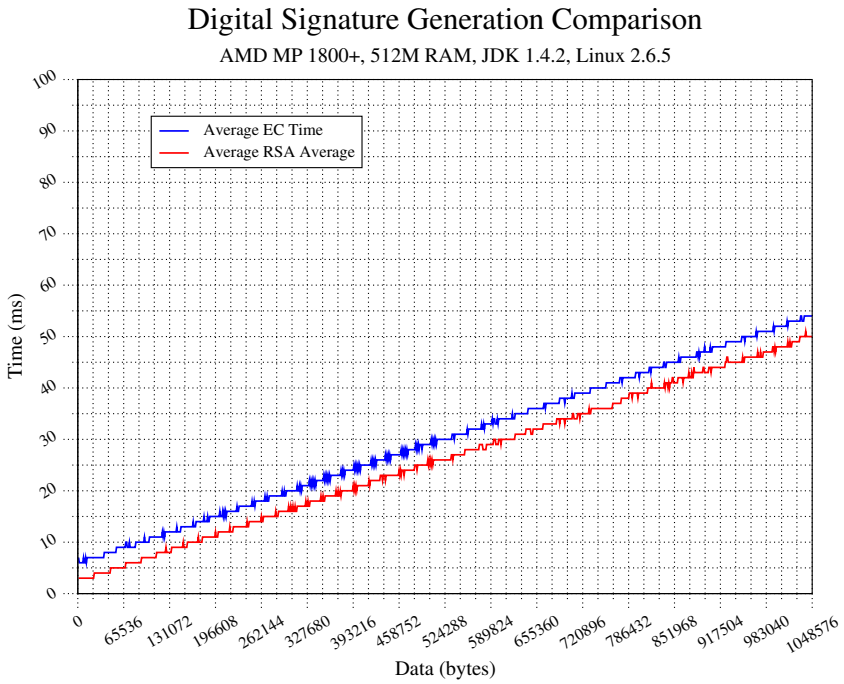

Figure 7.20: Signature Timing Comparison - Avg (AMD)

Chapter 8

Future Work

8.1 Authentication and Access Control

As mentioned in the introduction, user authentication and access control is somewhat outside the scope of this thesis. However, in an effort to describe a complete security framework, a brief mention will be made here.

It is clear that in a serverless environment, reliance upon a database of any form of usernames and passwords is prohibited. This database would need to be replicated to every node and painfully kept current. It is very clear that password-based authentication simply will not suffice in this paradigm.

Digital certificates, on the other hand, would provide a more robust and scalable system for user authentication. A user would generate a certificate, and have the appropriate third parties (such as departments, organizations, etc.) sign this certificate, building a hierarchy and web of authentication as needed. When joining a group, the user could present this certificate, and the other nodes could quickly and safely asertain the validity of the users credentials. The third party signatures that were gathered would serve to provide an access control list feature; if a particular network group is established requiring certain credentials and a certificate is presented without that the proper signatures, the certificate is rejected and the user is unable to join. This form of access control and user authentication is similar to techniques used in popular web server software, and is further described in [34].

Of course, this is not without technical difficulties as well. These certificates would need to be relatively short-lived to maintain correct access control. Resouce changes (ie people moving between departments or projects) would need to be reflected in these certificate signing chains. Revocation lists and proper signing sequences would need to be maintained, and each device would need to be synchronized with these changes. While somewhat better than the user-password database, there are many commonalities.

8.2 Key Agreement

There are clearly several choices for the key agreement system employed by a tool such as this research, each with strengths and weaknesses.

Tree-based

1. Structure of n group members has $n - 1$ intermediate values for a total of $2n - 1$ total keys. Each of these keys must be deserialized from bytes into key objects before agreement begins, regardless of which keys are required for agreement.
2. Determination of group key for any member is computed in $\log_2 n$ agreement rounds (from members' position in the tree to the tree root), and with n members in the group, this becomes $n \log_2 n$ total group-wide agreement rounds.

Array-based/Linear

1. Structure of n group members has no intermediate keys.
2. Determination of final group key requires n rounds for each node with a final calculation to determine members' share of final key, resulting in $(n^2) + 1$ total group-wide calculation rounds.

Depending on primary resource concerns, choice of key structure is core to overall performance. A more network-aware protocol should likely use a linear model to optimize network utilization. On the other hand, a protocol where network latency and bandwidth are bountiful and computational power is limited, a more two dimensional approach would be more appropriate to limit the number of agreement rounds required by each node to compute the final group key.

A tree-based design was chosen for this thesis work as the target platform for the framework is small devices. In general, network access and communication requires little computational effort, whereas key generation and computation is relatively intensive. Minimizing computation at the expense of increased network utilization fits well into the small device requirement and ability set.

8.3 Protocol Features

The stability of the tree-based agreement structure is solid for single-member operations, as demonstrated by numerous performance and functionality tests. However, multi-member operations (group merge or partition) place hurdles in our path.

A feature of the protocol, other work such as [19] has been done to alleviate these issues with respectable results.

Additionally, this thesis work does not include provisions for dealing with silent member leave events due to proximity restrictions or battery life.

8.4 Multi-MTU Message Transfer Techniques

In a TCP-based network setting, reliability and ordering issues are resolved by transparent protocols such as sliding window. These protocols are conversational — one party will send a message or messages, and the receiver will respond in an affirmational manner.

In a multicast or broadcast setting these techniques are entirely the wrong approach. A message broken into m sub-messages, each one MTU long, sent to n group members results in $m * n$ replies; more if any messages need to be resent due to network failures. This is clearly not a scalable solution.

8.4.1 Java's JRMS Package

Work has been done, as reflected in the Java Reliable Multicast Service (JRMS) package [28], to make multicast communications reliable. It would be entirely possible to use the LRMP (Light-Weight Reliable Multicast Protocol) profile from JRMS as a communication system framework for this codebase.

8.4.2 Resend with ACK-NACK

Another technique investigated briefly during the course of this work was one of repeated transmissions. It was observed that with twenty percent network delivery failure rate, resending a message three times ensured with a great degree of certainty that all members received the message. It is clear, however, that sending every message three times results in a three-fold inflation of network requirements. Lossier network systems requiring additional resends would only worsen these requirements, forming a cyclic dependency.

A manageable solution for multi-MTU messages is a combination resend-ACK/NACK system. Single MTU messages could simply be rebroadcast multiple times. Multiple MTU messages, on the other hand, could be reliably transfered using a resend threshold followed by a sequence of NACK-driven transfers.

Consider the following scenario. A message, *msg*, is 10 MTU long, with a resend threshold and resend counter values of three, and the aforementioned network environment with eighty percent delivery rate of UDP datagrams. Submessages of MTU-length

each are each sequenced as submessages of a larger message; this ensures proper reassembly ordering upon receipt. Submessage zero has some record of the total number of submessages needed to represent the complete message.

Submessages zero through two (the first three) are sent serially three times. (This could be done in a 0-0-0, 1-1-1, 2-2-2 pattern or 0-1-2, 0-1-2, 0-1-2 pattern. It is likely, however, the latter pattern would afford a higher level of group-wide receipt of the whole message sequence as there are three distinct time periods in which receiption of the entire message sequence is possible.) This ensures, with reasonable certainty, that all members will receive *some part* of this message. Receipt of any part of these first three triggers a response acknowledging receipt of these messages. Each node would respond to the sender to this affect, and resends would be performed to be sure each node has these three leading submessages. Now that all nodes know the total submessage count, these remaining blocks can be bundled and retransmitted. This process continues until all messages are cleanly transferred and not answered with NACK messages. This is an approximation of TCP's best-effort transfer approach, simply taken from a reverse perspective. Any node missing a submessage or range of submessages would simply reply to the sender to this effect; these NACKs could be collected over a short time period and unique requests be retransferred.

This attempts to find a harmony between network communication and reliability constructs — reliable transfer of submessages needs to be ensured to guarantee full and complete reconstruction of larger messages, yet must keep network communication to a minimum for efficiency and speed purposes.

This technique works equally well for point-to-point transfers, such as initial key tree delivery to a joining member, and broadcast transfers, such as setting a new key tree for the entire group.

The number of submessages resent and the number of times these messages are resent would each need to be configurable. Some analysis of live-network performance could be done to modify these control points to provide more real-time performance improvements. These controls are related to the flow control and collision control aspects of networking; tuning these approaches TCP-like governances.

Some foundational work was done along these lines to support multi-MTU messages during the key distribution phase of the protocol with some performnce and applicability testing. This was later abandoned in the interest of time and research scope.

8.4.3 Stream Control Transmission Protocol

Defined in RFC 2960 [36] and enhanced by RFC's 3309 [37] and 3436 [20], Stream Control Transmission Protocol (SCTP) could provide an answer to multicast multi-MTU

messaging in a group setting. SCTP provides reliable transmission of messages over an unreliable, connectionless network link.

8.5 Key Distribution

Once a key has been decided upon (leaving agreement aside for a moment) the next step is distribution of the key. In a contributory system where each member of the group influences the final key in some way, each member really only needs a small value of the final key – specifically their modified contribution. This is different from a server-based key management system where each node would need the full key, and other contributory systems where different requirements apply. Linear agreement, for example, would require each node possess the entire chain, whereas hypercubic agreement would only require a subset of the whole cube.

Trees provide a road map for each node to determine which other pieces of the tree are required. In general terms, if nodes were not simply at the leaves but within the tree structure as well, any intermediate node would simply need to send $1 - numberOfImmediateChildren$ values to each child (clearly the correct block to the correct subtree). This continued division and uncoupling of values from the other blocks of the key provides a framework where all nodes get all the messages they require to build the key, and only these messages. A side effect of this technique is that if a subset of nodes do not receive the new key, they each send NACK messages. The root node would then package the corresponding values into one message and broadcast this message to the entire group. Those not needing or expecting a rekeying message would simply ignore this broadcast.

It is also possible that agreement paradigms do not distribute the entire key management structure to all group members. In this case, it is advisable to distribute the current structure to or maintain recent versions of the structure with several members. This promotes general group stability and availability. The redundant nature of distribution in this sense allows for a higher rate of failure survivability.

8.6 Improved Network Performace

A goal of computing in general is improving performance across the board. We can improve computation by using efficient algorithms, but network communication and utilization provides a second area of opportunity for performance enhancement.

8.6.1 Key Improvements

Analysis of the keys that are transported during the protocol phases reveals an interesting optimization opportunity. Each key transports a great deal of the key generation parameter information. While this might be important for blind key agreement techniques where each party does not know the parameters, this environment requires parameters be public. Key material could be separated from the parameter data and transported for a simple optimization. This would require object construction on the receivers side for each key, but this is, in a sense, already occuring.

8.6.2 Compression

Additionally, an excellent way to improve network performance is a decrease in network traffic. If a group is willing to take the additional computational impact, all messages could easily be compressed using a high-speed compression routine. Simply reducing the data on the network will improve overall network performance.

For implementation and integration purposes, compression could easily be implemented as a modular `edu.rit.m2mp.security.Channel` class and utilized in any program harnessing the M2MI/M2MP framework. This would allow transparent compression and decompression of messages as each message passes through the Channel.

Part IV

Appendices

Appendix A

Design and Specification

A.1 Message Hierarchy

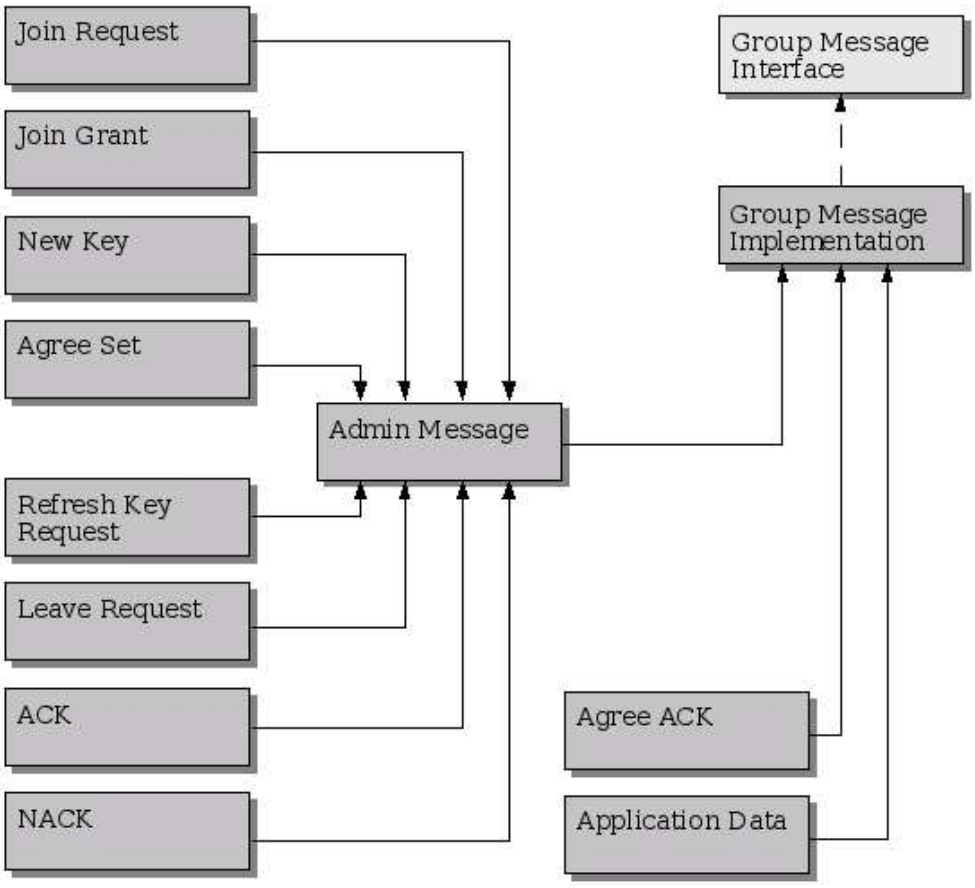

Figure A.1: Message Structure Heirarcy

Figure A.1 clearly shows that all messages are of type `GroupMessage`, and all extend the `groupMessageImpl` object. This provides a uniform pattern for serialization and

deserialization at both ends of communication. The methods in each message object, then, are minimal and provide only very specific operations. The `JoinRequest` and `JoinGrant` messages, for example, have functionality to correctly unpack and evaluate key parameters.

The vast majority of these objects extend `AdminMessage`; this was done in an effort to quickly multiplex messages based initially on object type. This also provides prompt tagging and identification of messages that could potentially modify the state and functioning of a group node or the entire group.

The `ACK` and `NACK` messages are not used in this implementation, but were implemented with the intention of utilization in a more reliable transmission system.

A.2 System Design

The `edu.rit.m2mp.security.utils` package is used in several places throughout the system, and is thus excluded from Figure A.2.

This shows the high-level relationships between the various modules comprising this work. Some functionality, such as the `KeyProxy` is acts as an inter-object channel, and eliminates the need for passing method calls through the reactor.

It is clear from the diagram that the reactor is the core of this system; all other modules work in direct conjunction with this centerpiece.

Table A.1 shows the correspondance of module to package name. The Communications and the Marshalling/Messages modules rely heavily on code from the `edu.rit.m2mp.security.utils` package. The `edu.rit.m2mp.security.signtures.hors` and `edu.rit.m2mp.security.keyagree` packages rely heavily on the `edu.rit.m2mp.security.keycommon` package for key processing and manipulation functionality.

Module Name	Package Name
Communications	`edu.rit.m2mp.security.comm`
Ciphers	`edu.rit.m2mp.security.crypto`
Marshalling/Messages	`edu.rit.m2mp.security.messages`
Signatures	`edu.rit.m2mp.security.signatures`
Agreement	`edu.rit.m2mp.security.keyagree`
Channel	`edu.rit.m2mp.security`

Table A.1: Module to Package Name Mapping

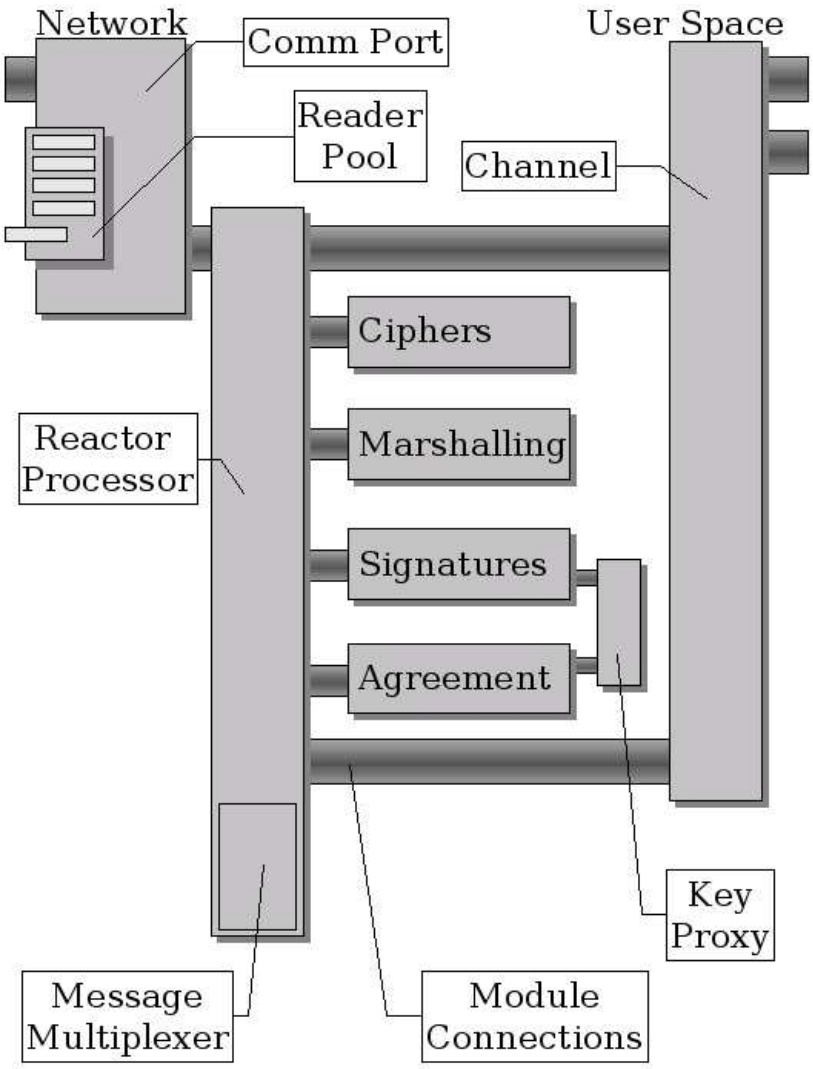

Figure A.2: High-level System Design

A.3 Network Design

Figure A.3 shows the message flow patterns for messages being read from and written to the netowrk. The module layout is the same as in Section A.2.

It is clear that messages read from the network must first pass through the decryption module, message deserialization, signature verification, and finally to the processing phase for complete validation. Messages written to the network are first properly wrapped, digitally signed, serialized, encrypted, and then written to the network.

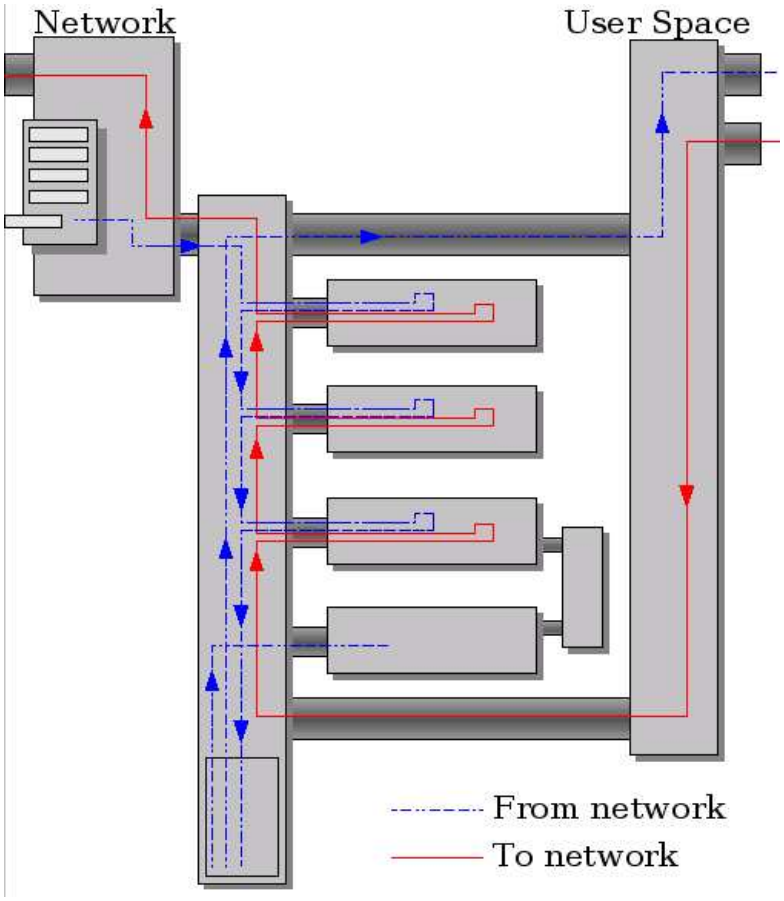

Figure A.3: Flow Pattern for Messages

A.4 Message Payload Format

All messages' payload region is by default a zero-length buffer with no specified format. For security-layer messages, the message type field (see Figure 4.1) identifies the intention of the message, and no payload is necessary. Exception to this rule are the `JoinRequest` and `JoinGrant` messages, outlined below.

A.4.1 JoinRequest

Start Byte	Byte Count	Description
0	1	Security mode value
1	> 1	Encoded algorithm parameters

Table A.2: JoinRequest Message Payload Structure

The *security mode value* field consists of the digital signature algorithm, cryptography algorithm, and the encryption mode values bitwise-OR'ed together.

A.4.2 JoinGrant

Start Byte	Byte Count	Description
0	1	Security mode value
1	1	Key agreement style
2	> 1	Encoded group algorithm parameters

Table A.3: JoinGrant Message Payload Structure

As with the `JoinRequest` message, the *Security mode value* field consists of the digital signature algorithm, cryptography algorithm, and the encryption mode values bitwise-OR'ed together.

Appendix B

Example Events

The following pages are transcript of three network nodes coming online and forming a group using this framework. Digital signatures are enabled, and 112-bit elliptic curve keys defined over a prime-order field are used for both generating the common encryption key and digital signature generation and validation.

In the transcript, messages sent are denoted with a '<–' prefix, and received mesages are prefixed with '–>'. Each message is reported in plaintext in the transcript, and several points of interest are breifly commented on.

The processing of each event is done simultaneously on different nodes; these commentaries are designed to help link common times in different nodes.

B.1 Three-Party Genesis

This example uses the elliptic curve-based key scheme, and public and private values are displayed when possible. These are verbose transcripts of a three-party group formation, showing the timeout and deadlock prevention at Genesis.

Node 1, Member ID 812015266

```
Launching discovery service... ... ... ... done
Discovery -- JoinRequest message is 136 bytes long
*Ping*

 -> (812015266)  Join Grant [Alone:Ready] from 1841218574
 -- Locked to joining sponsor '1841218574'
  Agreement -- Parameter algorithm: 'EC-GF(P)' - 1
  Agreement -- Obtained AlgorithmParameters object [ECDSA/1]
  Agreement -- Initialized object with encoded form
  Agreement -- Saved Algorithm Parameters
>> Generating local keypair ('EC-GF(P)')... ... ... ... ... setting in tree ... ... done
 <- New Key [Tree Wait:TreeWait] Set Key Type to 'EC-GF(P)'
Sending 181-byte 'New Key' to 812015266

 -> (812015266)  Join Request [??:??] from -700664168
```

As this node is currently joining to another node, this request is ignored.

```
 -> (812015266)  Tree Set Done [Tree Wait:Tree Wait] from 1841218574
  Agreement -- Instantiating KeyFactory object ['EC-GF(P)'/EC: true] ...[ECDH] done
  Agreement -- Generating Public Key... ...((82117b1573da23ff52d91e800b58, 834d7b6882a7fdb32e9771dc05be))... done
  Agreement -- Generating Public Key... ...((65bfd7905186bd6f8e136e1f02a5, 27fcebac500b59483785d2f437c))... done
  Agreement -- Generating Public Key... ...((da1db9afe6654ab53ad2e3fef331, aa38cb36dc81b9db009f676fe300))... done
  Member keypair: Priv: 105079632827265449327672226639063 3 Pub: (da1db9afe6654ab53ad2e3fef331, aa38cb36dc81b9db009f676fe300)
1) (812015266) agreePriv: 105079632827265449327672226639063 3
   (1841218574) agreePub: (65bfd7905186bd6f8e136e1f02a5, 27fcebac500b59483785d2f437c)
  Agreement -- Doing pair-wise agreement ... ... (Null inter. key: false)... building full
      [002](812015266) Pp
[000](INT) Pp
      [001](1841218574) P-
Null check -- private: false public: false
Generated (-):
```

```
0) Private: 93398083509384795420827459852329 3  Public: (82117b1573da23ff52d91e800b58, 834d7b6882a7fdb32e9771dc05be)

*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
Done with tree regeneration
      [002](812015266) Pp
[000](INT) Pp
      [001](1841218574) P-

  Agreement (Regenerate) -- New Group Key: 933980835093847954208274598523293
 <- TreeAck (New Membership Established) [Member:Ready] ()
Sending signed 56-byte 'Tree ACK' to 812015266
```

A new two-party group has been formed with a common private key. See page 90 for related state in other node.

```
 -> (812015266)  Join Request [??:??] from -700664168

 -> (812015266)  Join Request [Member:Ready] from -700664168
 <- Join Grant [Member:Key Wait] to -700664168 with key type 'EC-GF(P)'/ECDSA
Sending signed 173-byte 'Join Grant' to 812015266

 -> (812015266)  New Key [Member:Key Wait] from -700664168
  Agreement -- Generating Public Key... ...((65a47e5ffd3d5a3038ebfafb8daa, 50329dc17cecf1d9c03e2a0c5c91))... done
  Agreement -- Adding '-700664168'; I'm the sponsor
>> Generating local keypair ('EC-GF(P)')... ... ... ... setting in tree ... ... done
  Member keypair: Priv: 832149759619182116166361727456452 Pub: (bfe5cb35c852648723f482792830, 7ddd4e6db45a5b27f4acc2c0ad51)
2) (812015266) agreePriv: 832149759619182116166361727456452
   (1841218574) agreePub: (65bfd7905186bd6f8e136e1f02a5, 27fcebac500b59483785d2f437c)
  Agreement -- Doing pair-wise agreement ... ... (Null inter. key: false)... building full
      [002](-700664168) P-
[000](INT) --
            [004](812015266) Pp
      [001](INT) Pp
            [003](1841218574) P-
Null check -- private: false public: false
```

```
Generated (+):
1) Private: 797376572626094420889632715207356  Public: (b60119ba8bd11f061cce7fe75ea6, 20687107b99377aa31a545cf2e86)
```

This shows that a new member is joining the group, and it was the responsibility of this node to compute the new group keying information. Three members now exist in the group (leaves of tree), with two intermediate key values.

```
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
1) (-2147483648) agreePriv: 797376572626094420889632715207356
   (-700664168) agreePub: (65a47e5ffd3d5a3038ebfafb8daa, 50329dc17cecf1d9c03e2a0c5c91)
  Agreement -- Doing pair-wise agreement ... ... (Null inter. key: false)... building full
      [002](-700664168) P-
[000](INT) Pp
            [004](812015266) Pp
      [001](INT) Pp
            [003](1841218574) P-
Null check -- private: false public: false
Generated (+):
0) Private: 3157289455532251061550979072749584  Public: (35bf0e81ae36d22af0b6da5f3da9, 1941d8f5f9ce24f8082fb92d57bc)

*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
Done with tree regeneration
      [002](-700664168) P-
[000](INT) Pp
            [004](812015266) Pp
      [001](INT) Pp
            [003](1841218574) P-

  Agreement -- New Group Key: 3157289455532251061550979072749584
MEMBER '-700664168' JOINED -- COMPUTED NEW KEY
 <- Tree Set [Member:Ready] ()
Sending signed 897-byte 'Tree Set Done' to 812015266

 -> (812015266)  Tree Ack [* * * *:Tree Send] from -700664168
```

Node 2, Member ID 1841218574

```
Launching discovery service... ... ... ... done
Discovery -- JoinRequest message is 136 bytes long
*Ping*

 -> (1841218574)  Join Request [??:??] from 812015266

 -> (1841218574)  Join Request [Alone:Ready] from 812015266

 -- 812015266 < 1841218574 -- locking partner id & initiating
>> Generating local keypair ('EC-GF(P)')... ... ... ... ... setting in tree ... ... done
Sending 137-byte 'Join Grant' to 1841218574

 -> (1841218574)  Join Request [??:??] from -700664168
```

As this node is currently joining to another node, this request is ignored.

```
 -> (1841218574)  New Key [Key Wait:Key Wait] from 812015266
  Agreement -- Instantiating KeyFactory object ['EC-GF(P)'/EC: true] ...[ECDH] done
  Agreement -- Generating Public Key... ...((da1db9afe6654ab53ad2e3fef331, aa38cb36dc81b9db009f676fe300))... done
  Agreement -- Adding '812015266'; I'm the sponsor
>> Generating local keypair ('EC-GF(P)')... ... ... ... setting in tree ... ... done
  Member keypair: Priv: 75992242814961082834059758814708 2 Pub: (65bfd7905186bd6f8e136e1f02a5, 27fcebac500b59483785d2f437c)
1) (1841218574) agreePriv: 7599224281496108283405975881470 82
   (812015266) agreePub: (da1db9afe6654ab53ad2e3fef331, aa38cb36dc81b9db009f676fe300)
  Agreement -- Doing pair-wise agreement ... ... (Null inter. key: false)... building full
      [002](812015266) P-
[000](INT) Pp
      [001](1841218574) Pp
Null check -- private: false public: false
Generated (+):
0) Private: 93398083509384795420827459852329 3  Public: (82117b1573da23ff52d91e800b58, 834d7b6882a7fdb32e9771dc05be)
```

```
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
Done with tree regeneration
      [002](812015266) P-
[000](INT) Pp
      [001](1841218574) Pp

  Agreement -- New Group Key: 933980835093847954208274598523293
MEMBER '812015266' JOINED -- COMPUTED NEW KEY
 <- Tree Set [Member:Ready] ()
Sending signed 562-byte 'Tree Set Done' to 1841218574

 -> (1841218574)  Tree Ack [* * * *:Tree Send] from 812015266
```

A new two-party group has been formed with a common private key. See page 87 for related state in other node.

```
 -> (1841218574)  Join Request [??:??] from -700664168

 -> (1841218574)  Join Request [Member:Ready] from -700664168
 <- Join Grant [Member:Key Wait] to -700664168 with key type 'EC-GF(P)'/ECDSA
Sending signed 173-byte 'Join Grant' to 1841218574

 -> (1841218574)  New Key [Member:Key Wait] from -700664168
  Agreement -- Generating Public Key... ...((65a47e5ffd3d5a3038ebfafb8daa, 50329dc17cecf1d9c03e2a0c5c91))... done
  Agreement -- Adding '-700664168'; I'm NOT the sponsor
AT THIS POINT, I SHOULD BE BLOCKED & WAITING FOR KEY NOTIFY/SET...
  Agreement -- Removing '-700664168'; I'm NOT the sponsor
AT THIS POINT, I SHOULD BE BLOCKED & WAITING FOR KEY NOTIFY/SET...
MEMBER '-700664168' JOINED -- WAITING FOR NEW KEY
State: 'Member' Signed: true
```

The third member is joining the group, and this node realizes it is not responsible for computing the new group key; it waits for notification (below) of the new key tree.

```
 -> (1841218574)  Tree Set Done [Member:Ready] from 812015266
  Agreement -- Generating Public Key... ...((35bf0e81ae36d22af0b6da5f3da9, 1941d8f5f9ce24f8082fb92d57bc))... done
  Agreement -- Generating Public Key... ...((b60119ba8bd11f061cce7fe75ea6, 20687107b99377aa31a545cf2e86))... done
  Agreement -- Generating Public Key... ...((65a47e5ffd3d5a3038ebfafb8daa, 50329dc17cecf1d9c03e2a0c5c91))... done
  Agreement -- Generating Public Key... ...((65bfd7905186bd6f8e136e1f02a5, 27fcebac500b59483785d2f437c))... done
  Agreement -- Generating Public Key... ...((bfe5cb35c852648723f482792830, 7ddd4e6db45a5b27f4acc2c0ad51))... done
  Member keypair: Priv: 759922428149610828340597588147082 Pub: (65bfd7905186bd6f8e136e1f02a5, 27fcebac500b59483785d2f437c)
2) (1841218574) agreePriv: 759922428149610828340597588147082
   (812015266) agreePub: (bfe5cb35c852648723f482792830, 7ddd4e6db45a5b27f4acc2c0ad51)
  Agreement -- Doing pair-wise agreement ... ... (Null inter. key: false)... building full
      [002](-700664168) P-
[000](INT) P-
            [004](812015266) P-
      [001](INT) Pp
            [003](1841218574) Pp
Null check -- private: false public: false
Generated (-):
1) Private: 797376572626094420889632715207356  Public: (b60119ba8bd11f061cce7fe75ea6, 20687107b99377aa31a545cf2e86)

*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
1) (-2147483648) agreePriv: 797376572626094420889632715207356
   (-700664168) agreePub: (65a47e5ffd3d5a3038ebfafb8daa, 50329dc17cecf1d9c03e2a0c5c91)
  Agreement -- Doing pair-wise agreement ... ... (Null inter. key: false)... building full
      [002](-700664168) P-
[000](INT) Pp
            [004](812015266) P-
      [001](INT) Pp
            [003](1841218574) Pp
Null check -- private: false public: false
Generated (-):
0) Private: 3157289455532251061550979072749584  Public: (35bf0e81ae36d22af0b6da5f3da9, 1941d8f5f9ce24f8082fb92d57bc)
```

```
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
Done with tree regeneration
      [002](-700664168) P-
[000](INT) Pp
            [004](812015266) P-
      [001](INT) Pp
            [003](1841218574) Pp

  Agreement (Regenerate) -- New Group Key: 3157289455532251061550979072749584
 -- (New Key Established) [Member:Ready] ()

 -> (1841218574)  Tree Ack [* * * *:Tree Send] from -700664168
```

Node 3, Member ID -700664168

```
Launching discovery service... ... ... ... done
Discovery -- JoinRequest message is 136 bytes long
*Ping*

 -> (-700664168)  Join Grant [Alone:Ready] from 1841218574
 -- Locked to joining sponsor '1841218574'
  Agreement -- Parameter algorithm: 'EC-GF(P)' - 1
  Agreement -- Obtained AlgorithmParameters object [ECDSA/1]
*Ping*
```

This marks a timeout has occurred. The first request was ignored by all members of the group (see pages 86 and 89), thus prompting a second probe.

```
  Agreement -- Initialized object with encoded form
  Agreement -- Saved Algorithm Parameters
>> Generating local keypair ('EC-GF(P)')... ... ... ... ... setting in tree ... ... done
 <- New Key [Tree Wait:TreeWait] Set Key Type to 'EC-GF(P)'
Sending 181-byte 'New Key' to -700664168
```

```
 -> (-700664168)  Tree Set Done [Tree Wait:Tree Wait] from 812015266
  Agreement -- Instantiating KeyFactory object ['EC-GF(P)'/EC: true] ...[ECDH] done
  Agreement -- Generating Public Key... ...((35bf0e81ae36d22af0b6da5f3da9, 1941d8f5f9ce24f8082fb92d57bc))... done
  Agreement -- Generating Public Key... ...((b60119ba8bd11f061cce7fe75ea6, 20687107b99377aa31a545cf2e86))... done
  Agreement -- Generating Public Key... ...((65a47e5ffd3d5a3038ebfafb8daa, 50329dc17cecf1d9c03e2a0c5c91))... done
  Agreement -- Generating Public Key... ...((65bfd7905186bd6f8e136e1f02a5, 27fcebac500b59483785d2f437c))... done
  Agreement -- Generating Public Key... ...((bfe5cb35c852648723f482792830, 7ddd4e6db45a5b27f4acc2c0ad51))... done
  Member keypair: Priv: 3091192612846610570814386186041749 Pub: (65a47e5ffd3d5a3038ebfafb8daa, 50329dc17cecf1d9c03e2a0c5c91)
1) (-700664168) agreePriv: 3091192612846610570814386186041749
   (-2147483648) agreePub: (b60119ba8bd11f061cce7fe75ea6, 20687107b99377aa31a545cf2e86)
  Agreement -- Doing pair-wise agreement ... ... (Null inter. key: false)... building full
      [002](-700664168) Pp
[000](INT) Pp
            [004](812015266) P-
      [001](INT) P-
            [003](1841218574) P-
Null check -- private: false public: false
Generated (-):
0) Private: 3157289455532251061550979072749584  Public: (35bf0e81ae36d22af0b6da5f3da9, 1941d8f5f9ce24f8082fb92d57bc)

*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
Done with tree regeneration
      [002](-700664168) Pp
[000](INT) Pp
            [004](812015266) P-
      [001](INT) P-
            [003](1841218574) P-

  Agreement (Regenerate) -- New Group Key: 3157289455532251061550979072749584
 <- TreeAck (New Membership Established) [Member:Ready] ()
Sending signed 56-byte 'Tree ACK' to -700664168
```

The corresponences between Nodes 1 and 2 shows that these formed the initial pairing, and generated share private key 933980835093847954208274598523293. Node 3 was initally admitted to the group by Node 1, who later realized they were not suitable for sponsorship – this session suffered a timeout, and resumed correctly with Nodes 2 and 3 cooperating to build and distribute the final key, where all three members share the value 315728945553225106155097907274958 4.

Table B.1 and Table B.2 show network utilization for the three-way genesis event.

Node 1 (Writing)

Message Type	Quantity	Size Each	Total
JoinRequest	1	136	136
NewKey	1	181	181
AgreeACK	1	56	56
JoinGrant	1	173	173
AgreeSetDone	1	897	897
			1443

Node 2 (Writing)

Message Type	Quantity	Size Each	Total
JoinRequest	1	136	136
AgreeSetDone	1	562	562
JoinGrant	1	173	173
			871

Node 3 (Writing)

Message Type	Quantity	Size Each	Total
JoinRequest	2	136	272
NewKey	1	181	181
AgreeACK	1	56	56
			509

Table B.1: Communications for 3-Party Genesis - Writing

Totaling the written byte count (Table B.1) gives 2823 sent; totaling the read count (Table B.2) gives 4057. This disparity is due entirely to the fact that the read count duplicates messages (ie. the `AgreeSetDone` and `JoinRequest`) that are broadcast to the entire group.

Node 1 (Reading)

Message Type	Quantity	Source Node	Size Each	Total
JoinGrant	1	2	137	137
JoinRequest	2	3	136	272
AgreeSetDone	1	2	562	562
NewKey	1	3	181	181
AgreeAck	1	3	56	56
			1208	

Node 2 (Reading)

Message Type	Quantity	Size Each	Total	
JoinRequest	1	1	136	136
JoinRequest	2	3	136	272
AgreeSetDone	1	1	897	897
NewKey	1	1	181	181
NewKey	1	3	181	181
AgreeAck	1	1	56	56
AgreeAck	1	3	56	56
				1779

Node 3 (Reading)

Message Type	Quantity	Size Each	Total	
JoinGrant	1	2	173	173
AgreeSetDone	1	1	897	897
				1070

Table B.2: Communications for 3-Party Genesis - Reading

B.2 Member Leave

This is a continuation of the previous section; one of the three members leaves the group. The other two then generate a fresh key.

Node 1, Member ID 812015266

```
Discovery -- Killed...
 <- Leave Request [Alone:Ready]
Sending signed 52-byte 'Leave Request' to 812015266

Closing Channel...
```

Node 2, Member ID 1841218574

```
State: 'Member' Signed: true
>> Processing leave request

 -> (1841218574)  Leave Request [Member:Ready] from 812015266
  Agreement -- Removing '812015266'; I'm NOT the sponsor
AT THIS POINT, I SHOULD BE BLOCKED & WAITING FOR KEY NOTIFY/SET...
MEMBER '812015266' REMOVED -- WAITING FOR NEW KEY
State: 'Member' Signed: true

 -> (1841218574)  Tree Set Done [Member:Ready] from -700664168
  Agreement -- Generating Public Key... ...((84bf00ca3f9d4fbc49107f768cae, 4f9ca08f46f54b64277b94701da4))... done
  Agreement -- Generating Public Key... ...((65bfd7905186bd6f8e136e1f02a5, 27fcebac500b59483785d2f437c))... done
  Agreement -- Generating Public Key... ...((7b8731588421d9475229973a4f3e, 235e2aa47024437967f51d5b140c))... done
  Member keypair: Priv: 75992242814961082834059758814708 2 Pub: (65bfd7905186bd6f8e136e1f02a5, 27fcebac500b59483785d2f437c)
1) (1841218574) agreePriv: 75992242814961082834059758814708 2
   (-700664168) agreePub: (7b8731588421d9475229973a4f3e, 235e2aa47024437967f51d5b140c)
  Agreement -- Doing pair-wise agreement ... ... (Null inter. key: false)... building full
      [002](-700664168) P-
[000](INT) Pp
      [001](1841218574) Pp
Null check -- private: false public: false
Generated (-):
0) Private: 24904190204228210064239771676527 70  Public: (84bf00ca3f9d4fbc49107f768cae, 4f9ca08f46f54b64277b94701da4)
```

```
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
Done with tree regeneration
      [002](-700664168) P-
[000](INT) Pp
      [001](1841218574) Pp

  Agreement (Regenerate) -- New Group Key: 2490419020422821006423977167652770
 -- (New Key Established) [Member:Ready] ()
```

Node 3, Member ID -700664168

```
State: 'Member' Signed: true
>> Processing leave request

 -> (-700664168)  Leave Request [Member:Ready] from 812015266
  Agreement -- Removing '812015266'; I'm the sponsor
>> Generating local keypair ('EC-GF(P)')... ... ... ... setting in tree ... ... done
  Member keypair: Priv: 3357167766165848466468042362884883 Pub: (7b8731588421d9475229973a4f3e, 235e2aa47024437967f51d5b140c)
1) (-700664168) agreePriv: 3357167766165848466468042362884883
   (1841218574) agreePub: (65bfd7905186bd6f8e136e1f02a5, 27fcebac500b59483785d2f437c)
  Agreement -- Doing pair-wise agreement ... ... (Null inter. key: false)... building full
      [002](-700664168) Pp
[000](INT) Pp
      [001](1841218574) P-
Null check -- private: false public: false
Generated (+):
0) Private: 2490419020422821006423977167652770  Public: (84bf00ca3f9d4fbc49107f768cae, 4f9ca08f46f54b64277b94701da4)

*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
Done with tree regeneration
      [002](-700664168) Pp
```

```
[000](INT) Pp
      [001](1841218574) P-

  Agreement -- New Group Key: 249041902042282100642397716765277O
MEMBER '812015266' REMOVED -- COMPUTED NEW KEY
 <- Tree Set [Member:Ready] ()
Sending signed 560-byte 'Tree Set Done' to -700664168
```

After Node 1 left the group, and Nodes 2 and 3 regenerated a common key of 249041902042282100642397716765277O.

Appendix C

HORS Keypair and Key Exchange

```
* Provider loaded cleanly (index 6)
* KeyPairGenerator loaded cleanly
  KeyPairGenerator Information:
        Algorithm: HORS Provider: HORS
* KeyPairGenerator loaded cleanly
  KeyPairGenerator Information:
        Algorithm: HORS Provider: HORS
* KeyPair generated cleanly
  Key Information:
        Public : Alg: HORS Fmt: X.509 HORS Public Key
    Parameters:
        hashID: 0x2
        l:      0x80
        k:      0x10
        t:      0xa0
        OID:    1.3.6.1.4.1.4447.1
HashID: 2
0) -4795f4bff07881489b434f4764e654d7fb3d280d       1) 63e06037c7dab39c91cabc0c9453a31506e71ca9
2) 43a412bce0d31c3d28da78a5d07cae36a8c876a6        3) -1efd40d72a06ea9443fa9f660be8dbaf316537c
```

```
 4) -37e5606899ee025374dc1d2e82f6ed38c21a0043
 6) 4b518cd9e6ea9c94a86b47e1a119621576ad50d3
 8) 15393557ea56a4c0a8b6f4c634b63047dcba610
10) -2e90d44cf86313e0c891783fba878acedcff5a00
12) -63374599d87332df1b71086c56ea5d1ccd800c64
14) -53bb1356c8a6df88b7e5dd00af934655435adaff
16) 517f58dd02be4f31c1e38100d5675ead1fa252ac
18) daa3c106e00d7ed43a2fae88ae42bf0aff58294
20) -66773d6eb6d311a5d85eca0a76a982433d26e90c
22) 5ef26f0d340aa260a1fc6e7d91533d801b9e2c8c
24) 6fffcbc5b5c0238b620327956496634c446e6407d
26) -4e923d7d0e8494b53bb21636f33ad9202c159759
28) -572552470f90c642bc6c54f2cd9aa10d5a54f5b
30) -53dbfa5e2a4b1fe556ee6c1831ab27a7eed91eb
32) -66f77749101fdfa56c35eeebf476906 4f53eaf2f
34) -6489d22f22f7c3110e404d85681661aaa021bc0b
36) 216831bc365dfe4828da40a73c20d50313bb17b6
38) -15b661fa7a2d0b9d644545acfab55ec1a5debd98
40) 164dc96380cdd533f544b17bb831cd40225ce03b
42) -4dc9d938851c3de905f8d5bd0cfd776a7946fee8
44) 39658ed5f3e48cdb882bc9536d2734c61e578eea
46) 5feca3295cf3ba5a92ed8874a9540c8e88d64120
48) 18d304676b01a613c5db8bd973cffcb63a7b2436
50) -5e6e1e73accb71a266a4f5e0851acaf8466ac4f0
52) -5bf0ea3e341157c3c6dbad39c01ab7190d6b37d1
54) 41141267c015d74cfc6bc0f66fe1d954c9ea7cf1b
56) -141a5bb1d451e7a89a699b738f7f054400dc0785
58) 662a58cea4ea4b754a8929931171e6c9547bbcaf
60) 4240cc872f3e93b5666b7cb3db58bab902ae673b
62) 39e8cd9718701d961a19e5ae6621b6f40a4c9e25
64) -563666dd1872fb9a90072020de6634da7b36c49f
66) -5e19700cdf5532004831a5744cd2f49f58e03aa
68) -7a5f129ad59608faf5b9452c702742b106594689
 5) -288ca6e8f82872e2af8672e0043cc876f725a318
 7) 746a27282ce4254649e909cc8dacfd29fe171c2a
 9) -3eaa0b93da9d8352164 58e1fe2c8f8ef7f17afbd
11) 2134156eee3a976f882ec12069a36ddda51c191c
13) -5b6acd7fa4c7a55867ef5626fbfa64a70633256b
15) 696ff6a17e804b70ddf8c3620ca2ae7440b4360e
17) 58de25876eec004264e35e59c1db2d8ac51551a5
19) -b242363ca0266f35daa5616c2c7ce4198e13d7d
21) 28bd1fb2332cf9a29c5e6a26ca2f0ddee61949fa
23) -3985a3cfd0a2ef6981ead261bdf8aa60e8e9651b
25) -2b38831dc47161777c734db09354f2aaf74434ec
27) -6b55a1c5a4daa0c55a1d9052c760d6a5e059274d
29) 6c72499e279331ebe4036477c68a071675eca8a1
31) -1064fd69451d255773fe9bd143221ddcd46f716f
33) 7710358ab76c6daf2ab453d1bf6185fba157adda
35) -3b6689c1c0081b187eec594d3d35e83e64699cc
37) -717a201960acfe5a049f8287c66ecc01d0ccb204
39) 5e3021df8504ee55715d064965f0d2e1270f8d24
41) 58e591b937c22282d4e588fbb3fd5f24f1cfc640
43) -208124318155745 0e747d54dbfc1632ba6a6bfa2
45) 3ad14d3524c49522e9ab79361d8e67dd0b471d5a
47) 768075b6badd1a9cb34b7d1d8c09b3d59af29fbb
49) -788175b990707f82ff22a00caada82558bee575b
51) 7216f61dca12f42d38fba374f758fa7a8f9adc76
53) -6c7834b348fe06b8cebf4b1dd4d376f2fa6b9dd4
55) 1065cc7e2d224b564dab8ba934ea3b413b9cad47
57) 7ae753b035d1a0ef603fc9d7bd8e55eed753f597
59) -782edf74b1d01038 9820c74d78ecdc4c85f1ea43
61) -30f760fcf9f15476cdfdec0f0da65bef682e7eaf
63) -5fd7908e7942f57421ebd94f3654aa94b41b036d
65) 3c39091b28fd0ba2b33176d996e93763338db65f
67) -777e6797a27ff9cb13f20a7a26fc3b853a844694
69) 3b13c05986ef502af6aca1c55b489e47012b7f92
```

70) -7b2bd4e11fc125b2f3c5bf643d7f0cf8448438e3
71) 57a4bed0647eccda1bdebfb2097179d2ff10126c
72) 1d5a00b5f3293987b24fb67347c87508e1a7c7e3
73) 1771293f04b9aab370ef8bc1a5302a69e4e3457b
74) -6b85d10ee501bf377f161e1709e77c95d5c2138c
75) 20e532cf7f772a01627cfa8b7bc36c7f720b212
76) 553eb14d1efaa193e3c1a6019521bc11f1de32de
77) -4769257e3d6900a776b69fac64596079906a7c5d
78) -6a90574e63f0a64cd2b78b2b11244949502e9462
79) 511e53e36cf4ad34f221e872d43758fab0d424b5
80) 56a205a37676068e212d45ddcc4a8a8dff4097d8
81) 2fa08506d05f1a20b32ee6d1ce58e85125d4a9cb
82) -39d10fd6d90699552957ea722454ef139d0325d4
83) 53f415c4aff8113e4f649494172da3809da73b0f
84) 3521946973eb2ff9c9e005cfa0ff6749103f318b
85) 49fe7c7b1035c52a1fd0ef292cf9d4fdbce27ed4
86) 29e56b21aee68d1b29ca30d7f1858ed7016a826a
87) 4464a76ae548c3ff19b233bed3bc2e9c6dc67ac2
88) 2960e36cfe87bc17ac603fc3cdb835f073df0c58
89) 469ec86a97b30c18b13d3a46738aae016b065e5b
90) 3942b1bf462819f60150aa7686dfbea84c8585ad
91) 1419e1efffa9ce050bff8e2b349f8eb5f23d9c5a
92) 70bdb1c14cf5cc760be62f418e0b9780ae953bbb
93) 69dff899949d6b1491cd06614753e48b7b6190cf
94) -7dc5b6991c563776283921b250313089f078cef3
95) -16ccaea6506c588e8adc6a864429e90965d9d86
96) -24e4f8e1eda857a4300a7e3a89b0ba4e7697a116
97) -30f6c8033facd32d714e7c1e90cd1e9bf419bd5b
98) 1bdfa9334bd0f28730f1570b3b9d1b1fd51bac43
99) 96f02d8ef6d2dcb5fb841e56f253780f1817173
100) 59eeaa3fcdb3879ab44ef2369d35eec881531cea
101) 43369ff63fa33aa9cced235c7b285c56c01c9579
102) -68c0406456e38e0bf98415862bce90cc53393a86
103) -76b018a655b095fbbb330f6dfeb778d0df06bfee
104) -36e89f1b2d5291bea20fa27c6906cf3645532414
105) 3a6a0159d53f85a9bfd3aeebbff2f1b62b34e0e7
106) 38a57e488941f3e1d8161fbd0271d76954692adc
107) 48cce375c05ed0b275dd018c33de3178490d7983
108) -281b7f2240862378f5895374d6003775e08fb36d
109) -49967f3c737a4c6b7793ee4d40e07ba70153e36d
110) 59ef895af9768d70cf1393dd10a9ad2af2719c3b
111) -45e258f82608f07346fe9ec82e2b6bc39835deee
112) -5f339e1bb95b8c386247d3b5bfaaf4fd46876922
113) -5023b39bc2c8e43d8c6d7952806365b164702634
114) 1200df78c59fb773000ee9be8a7c09709f088ace
115) -2cf8413f2e7ea14d7bcefc7415d092583686e6b
116) 6942811c65890ca2b6f42846506a2a92e8095416
117) 5f397aa55bb8136fd3f115afcdd7d1a231718ddf
118) 5cf5fc8a05a650dbdcf2dd8d5d269a45fa1e16c5
119) 275371f394668c63ef8d3964024cd84f610c94a3
120) 235740cab01b446d6b16e62fa10936687b006e0
121) 1434d2cfb171fb3759a235b7fbffe6f049d9b09f
122) 77ed36fb9638a0df957175937559122afc30c48b
123) 6bdbcec4f3b28ba42f3a380d7e5bfd70bbf81c2e
124) 77f285b2e201aaecaf34bc2885395c74baf0c64a
125) -3b6d839e31db27f2c1b2f62fd4b24aab2a69b2cf
126) 11d65394f3d33201cae8c4d5ba711dd86831cb95
127) ae3e49f44b4f517d76152237c37491f585b394b
128) 3bdd09f9c4f3af5b63460f00370b029ca1cdc41f
129) 66e9167e844a19cd70fa2f3e13685161e8f31339
130) -6a8fce8abc5a53e57e2ea722df7326292b408c01
131) -34989b263d27ffee6536548b3967b013a29825bb
132) 47aa31e06f4cb013e46d3ed3658358d791627b82
133) -47a540e187016b064ad470e1055ad0830238e409
134) 170369967f362948ed1502a94cf9326e3d7e4b5e
135) 582ca45b4351a007d697da4328537345ce2226d3

```
136) -5a26585a92e22bb2799b25197db02658811fcaa7d  137) -12d06ce6ab31ccc91671d730986045149706 4b27
138) 44df7cfb2b8a9f0000332a1e3d04e60227b866e9  139) -57f806d910c728fb8ae365fc4bd7079ecd246b65
140) -79437f1a2dbc1570a881d6495be8cb75fec8deef  141) -3eb61fbb127ff59c23a93eb12e8b004471a08474
142) 7b4d8ae78ccada0645f7bf09318323b50f7ad41c  143) 5f20e1f82bdf9ae9ea7000e834618d79099ec577
144) -7968eff810872fe2817a82f3345b8c2186ac9ce9  145) -39dbad1de7918e43961120920ece13f971605b52
146) -617e2b9b3462b9eb2e37d158d5a642996157421f  147) 7818107106f83c02a1254e0f782bfaa931e4b9b2
148) -2ebeafa97c62580b8cf3b255fd1aa909ce0756af  149) 67ae7b21ea32532d69f97d4b07cdb7cade820b64
150) 5795a2274c1c35dfc5d4ceea2f5dd0a1dbf808e6  151) 30308ebe4f3b9b3d82280e4da1a753988cee0fcf
152) 55781c3a37fe6b0bddd95d7e909b598826eb613d  153) 4ea03021f3af2c775da61892225b399f22f9585d
154) 7d6cc8eceb2b0e034cb90a790f8aa52d6254039a  155) -7ca51ffab1edb5374b2042e348509d94b49a0b27
156) -3b4053db9740557945182 3ddfa8c5caaa896ae9d  157) -103f86b495ae030dc8a4c64a9839e7d39d841b91
158) -2ffa0ea490475908547 0d2f9a327508659ddb8f6  159) -1421392260fc7beab407d13b46767c3cf4f6a860
     Private: Alg: HORS Fmt: PKCS#8 HORS Public Key
     Parameters:
          hashID: 0x2
               l:      0x80
               k:      0x10
               t:      0xa0
          OID:    1.3.6.1.4.1.4447.1
HashID: 2
0) -4795f4bff07881489b434f4764e654d7fb3d280d  1) 63e06037c7dab39c91cabc0c9453a31506e71ca9
2) 43a412bce0d31c3d28da78a5d07cae36a8c876a6  3) -1efd40d72a06ea9443fa9f660be8dbaf316537c
4) -37e5606899ee025374dc1d2e82f6ed38c21a0043  5) -288ca6e8f82872e2af8672e0043cc876f725a318
6) 4b518cd9e6ea9c94a86b47e1a119621576ad50d3  7) 746a27282ce4254649e909cc8dacfd29fe171c2a
8) 15393557ea56a4c0a8b6f4c634b63047dcba610  9) -3eaa0b93da9d8352164 58e1fe2c8f8ef7f17afbd
10) -2e90d44cf86313e0c891783fba878acedcff5a00  11) 2134156eee3a976f882ec12069a36ddda51c191c
12) -63374599d87332df1b71086c56ea5d1ccd800c64  13) -5b6acd7fa4c7a55867ef5626fbfa64a70633256b
14) -53bb1356c8a6df88b7e5dd00af934655435adaff  15) 696ff6a17e804b70ddf8c3620ca2ae7440b4360e
16) 517f58dd02be4f31c1e38100d5675ead1fa252ac  17) 58de25876eec004264e35e59c1db2d8ac51551a5
18) daa3c106e00d7ed43a2fae88ae42bf0aff58294  19) -b242363ca0266f35daa5616c2c7ce4198e13d7d
20) -66773d6eb6d311a5d85eca0a76a982433d26e90c  21) 28bd1fb2332cf9a29c5e6a26ca2f0ddee61949fa
22) 5ef26f0d340aa260a1fc6e7d91533d801b9e2c8c  23) -3985a3cfd0a2ef6981ead261bdf8aa60e8e9651b
24) 6fffcbc5b5c0238b6203279564 9634c446e6407d  25) -2b38831dc47161777c734db09354f2aaf74434ec
```

26) -4e923d7d0e8494b53bb21636f33ad9202c159759
27) -6b55a1c5a4daa0c55a1d9052c760d6a5e059274d
28) -572552470f90c642bc6c54f2cd9aa10d5a54f5b
29) 6c72499e279331ebe4036477c68a071675eca8a1
30) -53dbfa5e2a4b1fe556ee6c1831ab27a7eed91eb
31) -1064fd69451d255773fe9bd143221ddcd46f716f
32) -66f77749101fdfa56c35eeebf4769064f53eaf2f
33) 7710358ab76c6daf2ab453d1bf6185fba157adda
34) -6489d22f22f7c3110e404d85681661aaa021bc0b
35) -3b6689c1c0081b187eec594d3d35e83e64699cc
36) 216831bc365dfe4828da40a73c20d50313bb17b6
37) -717a201960acfe5a049f8287c66ecc01d0ccb204
38) -15b661fa7a2d0b9d644545acfab55ec1a5debd98
39) 5e3021df8504ee55715d064965f0d2e1270f8d24
40) 164dc96380cdd533f544b17bb831cd40225ce03b
41) 58e591b937c22282d4e588fbb3fd5f24f1cfc640
42) -4dc9d938851c3de905f8d5bd0cfd776a7946fee8
43) -2081243181557450e747d54dbfc1632ba6a6bfa2
44) 39658ed5f3e48cdb882bc9536d2734c61e578eea
45) 3ad14d3524c49522e9ab79361d8e67dd0b471d5a
46) 5feca3295cf3ba5a92ed8874a9540c8e88d64120
47) 768075b6badd1a9cb34b7d1d8c09b3d59af29fbb
48) 18d304676b01a613c5db8bd973cffcb63a7b2436
49) -788175b990707f82ff22a00caada82558bee575b
50) -5e6e1e73accb71a266a4f5e0851acaf8466ac4f0
51) 7216f61dca12f42d38fba374f758fa7a8f9adc76
52) -5bf0ea3e341157c3c6dbad39c01ab7190d6b37d1
53) -6c7834b348fe06b8cebf4b1dd4d376f2fa6b9dd4
54) 41141267c015d74cfc6bc0f6fe1d954c9ea7cf1b
55) 1065cc7e2d224b564dab8ba934ea3b413b9cad47
56) -141a5bb1d451e7a89a699b738f7f054400dc0785
57) 7ae753b035d1a0ef603fc9d7bd8e55eed753f597
58) 662a58cea4ea4b754a8929931171e6c9547bbcaf
59) -782edf74b1d010389820c74d78ecdc4c85f1ea43
60) 4240cc872f3e93b5666b7cb3db58bab902ae673b
61) -30f760fcf9f15476cdfdec0f0da65bef682e7eaf
62) 39e8cd9718701d961a19e5ae6621b6f40a4c9e25
63) -5fd7908e7942f57421ebd94f3654aa94b41b036d
64) -563666dd1872fb9a90072020de6634da7b36c49f
65) 3c39091b28fd0ba2b33176d996e93763338db65f
66) -5e19700cdf55320048831a5744cd2f49f58e03aa
67) -777e6797a27ff9cb13f20a7a26fc3b853a844694
68) -7a5f129ad59608faf5b9452c702742b106594689
69) 3b13c05986ef502af6aca1c55b489e47012b7f92
70) -7b2bd4e11fc125b2f3c5bf643d7f0cf8448438e3
71) 57a4bed0647eccda1bdebfb2097179d2ff10126c
72) 1d5a00b5f3293987b24fb67347c87508e1a7c7e3
73) 1771293f04b9aab370ef8bc1a5302a69e4e3457b
74) -6b85d10ee501bf377f161e1709e77c95d5c2138c
75) 20e532cf7f772a01627cfa8b7bc36c7f720b212
76) 553eb14d1efaa193e3c1a6019521bc11f1de32de
77) -4769257e3d6900a776b69fac64596079906a7c5d
78) -6a90574e63f0a64cd2b78b2b11244949502e9462
79) 511e53e36cf4ad34f221e872d43758fab0d424b5
80) 56a205a37676068e212d45ddcc4a8a8dff4097d8
81) 2fa08506d05f1a20b32ee6d1ce58e85125d4a9cb
82) -39d10fd6d90699552957ea722454ef139d0325d4
83) 53f415c4aff8113e4f649494172da3809da73b0f
84) 3521946973eb2ff9c9e005cfa0ff6749103f318b
85) 49fe7c7b1035c52a1fd0ef292cf9d4fdbce27ed4
86) 29e56b21aee68d1b29ca30d7f1858ed7016a826a
87) 4464a76ae548c3ff19b233bed3bc2e9c6dc67ac2
88) 2960e36cfe87bc17ac603fc3cdb835f073df0c58
89) 469ec86a97b30c18b13d3a46738aae016b065e5b
90) 3942b1bf462819f60150aa7686dfbea84c8585ad
91) 1419e1efffa9ce050bff8e2b349f8eb5f23d9c5a

92) 70bdb1c14cf5cc760be62f418e0b9780ae953bbb
94) -7dc5b6991c563776283921b2503130 89f078cef3
96) -24e4f8e1eda857a4300a7e3a89b0ba4e7697a116
98) 1bdfa9334bd0f28730f1570b3b9d1b1fd51bac43
100) 59eeaa3fcdb3879ab44ef2369d35eec881531cea
102) -68c0406456e38e0bf98415862bce90cc53393a86
104) -36e89f1b2d5291bea20fa27c6906cf3645532414
106) 38a57e488941f3e1d8161fbd0271d76954692adc
108) -281b7f224086237 8f5895374d6003775e08fb36d
110) 59ef895af9768d70cf1393dd10a9ad2af2719c3b
112) -5f339e1bb95b8c386247d3b5bfaaf4fd46876922
114) 1200df78c59fb773000ee9be8a7c09709f088ace
116) 6942811c65890ca2b6f42846506a2a92e8095416
118) 5cf5fc8a05a650dbdcf2dd8d5d269a45fa1e16c5
120) 235740cab01b446d6b16e62fa10936687b006e0
122) 77ed36fb9638a0df95717593755 9122afc30c48b
124) 77f285b2e201aaecaf34bc2885395c74baf0c64a
126) 11d65394f3d33201cae8c4d5ba711dd86831cb95
128) 3bdd09f9c4f3af5b63460f00370b029ca1cdc41f
130) -6a8fce8abc5a53e57e2ea722df7326292b408c01
132) 47aa31e06f4cb013e46d3ed3658358d791627b82
134) 170369967f362948ed1502a94cf9326e3d7e4b5e
136) -5a26585a92e22bb2799b25197db0265811fcaa7d
138) 44df7cfb2b8a9f0000332a1e3d04e60227b866e9
140) -79437f1a2dbc1570a881d6495be8cb75fec8deef
142) 7b4d8ae78ccada0645f7bf09318323b50f7ad41c
144) -7968eff810872fe2817a82f3345b8c2186ac9ce9
146) -617e2b9b3462b9eb2e37d158d5a642996157421f
148) -2ebeafa97c62580b8cf3b255fd1aa909ce0756af
150) 5795a2274c1c35dfc5d4ceea2f5dd0a1dbf808e6
152) 55781c3a37fe6b0bddd95d7e909b598826eb613d
154) 7d6cc8eceb2b0e034cb90a790f8aa52d6254039a
156) -3b4053db9740557945182 3ddfa8c5caaa896ae9d
93) 69dff899949d6b1491cd06614753e48b7b6190cf
95) -16ccaea6506c588e8adc6a864429e90965d9d86
97) -30f6c8033facd32d714e7c1e90cd1e9bf419bd5b
99) 96f02d8ef6d2dcb5fb841e56f253780f1817173
101) 43369ff63fa33aa9cced235c7b285c56c01c9579
103) -76b018a655b095fbbb330f6dfeb778d0df06bfee
105) 3a6a0159d53f85a9bfd3aeebbff2f1b62b34e0e7
107) 48cce375c05ed0b275dd018c33de3178490d7983
109) -49967f3c737a4c6b7793ee4d40e07ba70153e36d
111) -45e258f82608f07346fe9ec82e2b6bc39835deee
113) -5023b39bc2c8e43d8c6d795280636 5b164702634
115) -2cf8413f2e7ea14d7bcefc7415d092583686e6b
117) 5f397aa55bb8136fd3f115afcdd7d1a231718ddf
119) 275371f394668c63ef8d3964024cd84f610c94a3
121) 1434d2cfb171fb3759a235b7fbffe6f049d9b09f
123) 6bdbcec4f3b28ba42f3a380d7e5bfd70bbf81c2e
125) -3b6d839e31db27f2c1b2f62fd4b24aab2a69b2cf
127) ae3e49f44b4f517d76152237c37491f585b394b
129) 66e9167e844a19cd70fa2f3e13685161e8f31339
131) -34989b263d27ffee6536548b3967b013a29825bb
133) -47a540e18701 6b064ad470e1055ad0830238e409
135) 582ca45b4351a007d697da4328537345ce2226d3
137) -12d06ce6ab31ccc91671d730986045149706 4b27
139) -57f806d910c728fb8ae365fc4bd7079ecd246b65
141) -3eb61fbb127ff59c23a93eb12e8b004471a08474
143) 5f20e1f82bdf9ae9ea7000e834618d79099ec577
145) -39dbad1de7918e4396112 0920ece13f971605b52
147) 781810710 6f83c02a1254e0f782bfaa931e4b9b2
149) 67ae7b21ea32532d69f97d4b07cdb7cade820b64
151) 30308ebe4f3b9b3d82280e4da1a753988cee0fcf
153) 4ea03021f3af2c775da61892225b399f22f9585d
155) -7ca51ffab1edb5374b20 42e348509d94b49a0b27
157) -103f86b495ae030dc8a4c64a9839e7d39d841b91

```
158) -2ffa0ea4904759085470d2f9a327508659ddb8f6   159) -1421392260fc7beab407d13b46767c3cf4f6a860

* KeyPair generated cleanly
  Key Information:
        Public : Alg: HORS Fmt: X.509 HORS Public Key
    Parameters:
        hashID: 0x2
        l:      0x80
        k:      0x10
        t:      0xa0
        OID:    1.3.6.1.4.1.4447.1
HashID: 2

        Private: Alg: HORS Fmt: PKCS#8 HORS Public Key
    Parameters:
        hashID: 0x2
        l:      0x80
        k:      0x10
        t:      0xa0
        OID:    1.3.6.1.4.1.4447.1
HashID: 2


* Public Key generated cleanly
  Key Information:
        Public : 1.3.6.1.4.1.4447.1 X.509
* Public Key generated cleanly
  Key Information:
        Public : 1.3.6.1.4.1.4447.1 X.509
Alice's Key was successfully reconstructed
Bob's Key was successfully reconstructed
```

```
TIMING PROFILE (milliseconds):
    [x] Provider load:                  876
    [x] Key Generator load (Alice)       29
    [x] Key Generator load (Bob):         1
    [x] Keypair Generation (Alice):     638
    [x] Keypair Generation (Bob)        100
    [x] Public Key Exchange:              0
    [x] Rebuild Alices Key:              55
    [x] Rebuild Bobs Key:                10
        TOTAL TIME:                    1709
```

Appendix D

Codebase Statistics

Table and statistics generated by JavaNCSS package, available at
<http://www.kclee.com/clemens/java/javancss/>.

D.1 Lines of Code per Package

Special note: The key encoding functionality was adapted from the Sun JDK source download via [29], and resides in the `edu.rit.m2mp.security.keycommon` package. Some additional functionality exists in the `edu.rit.m2mp.security.keyagree` package for proper reconstruction of public and private keys

This is due to the need for key encoding support at a lower level than available through the `java.*` packages, and to compensate for the instability of the `sun.*` and `com.sun.*` packages. It was observed that direct dependence upon these 'hidden' packages proved fatal when moving between JDK versions 1.4.1 and 1.4.2.

D.2 Lines of Code per Java File

The statistics in Table D.2 are a partial report, trimmed for space considerations.

```
Thu, Apr 01, 2004  16:04:45 America/New_York

 Classes Methods   NCSS  Javadocs Package
      14     154   1619       158 edu.rit.m2mp.security
       2      24    342        26 edu.rit.m2mp.security.comm
       1       3    193         0 edu.rit.m2mp.security.comm.test
       1      15    192        16 edu.rit.m2mp.security.crypto
       1       4     61         5 edu.rit.m2mp.security.handlers
       1       5     40         0 edu.rit.m2mp.security.handlers.test
       9     149   1323       138 edu.rit.m2mp.security.keyagree
       6      41    458         2 edu.rit.m2mp.security.keyagree.test
      10     185   1363       154 edu.rit.m2mp.security.keycommon
      15     107    551       113 edu.rit.m2mp.security.messages
       2      13    163        13 edu.rit.m2mp.security.signatures
      12     144   1280       139 edu.rit.m2mp.security.signatures.hors
       5       9     26        14 edu.rit.m2mp.security.signatures.hors.interfaces
       3      17     75        20 edu.rit.m2mp.security.signatures.hors.spec
       4      39    858         2 edu.rit.m2mp.security.signatures.hors.test
       3      20    118        27 edu.rit.m2mp.security.utils
-------- ------- ------ ---------
      89     929   8662       827 Total

 Packages   Classes Functions      NCSS  Javadocs | per
------------------------------------------------------------
    16.00     89.00    929.00   8662.00    827.00 | Project
              5.56     58.06    541.38     51.69 | Package
                       10.44     97.33      9.29 | Class
                                  9.32      0.89 | Function
```

Table D.1: Codebase Statistics – Lines of Code per Package

Thu, Apr 01, 2004 16:04:45 America/New_York

NCSS	Methods	Class
184	3	edu.rit.m2mp.security.comm.test.CommPortTest
294	21	edu.rit.m2mp.security.comm.CommPort
30	3	edu.rit.m2mp.security.comm.Reader
2	1	edu.rit.m2mp.security.keycommon.DerEncoder
179	14	edu.rit.m2mp.security.keycommon.DerIndefLenConverter
171	17	edu.rit.m2mp.security.keycommon.DerInputBuffer
181	37	edu.rit.m2mp.security.keycommon.DerInputStream
173	34	edu.rit.m2mp.security.keycommon.DerOutputStream
349	50	edu.rit.m2mp.security.keycommon.DerValue
158	15	edu.rit.m2mp.security.keycommon.ObjectIdentifier
89	15	edu.rit.m2mp.security.keycommon.BitArray
10	1	edu.rit.m2mp.security.keycommon.ByteArrayLexOrder
5	1	edu.rit.m2mp.security.keycommon.ByteArrayTagOrder
184	15	edu.rit.m2mp.security.crypto.CryptoEngine
34	5	edu.rit.m2mp.security.handlers.test.DiscoveryTest
52	4	edu.rit.m2mp.security.handlers.Discovery
17	2	edu.rit.m2mp.security.keyagree.test.Test
170	13	edu.rit.m2mp.security.keyagree.test.KeySizeTest
40	6	edu.rit.m2mp.security.keyagree.test.AgreementLoaderTest
37	6	edu.rit.m2mp.security.keyagree.test.KeyDistroInvestigation
53	5	edu.rit.m2mp.security.keyagree.test.ParameterCycleTest
107	9	edu.rit.m2mp.security.keyagree.test.KeyAgreementTest
238	24	edu.rit.m2mp.security.keyagree.AgreementEngine
122	12	edu.rit.m2mp.security.keyagree.DHPrivateKey
111	12	edu.rit.m2mp.security.keyagree.DHPublicKey
267	31	edu.rit.m2mp.security.keyagree.AgreementImpl
128	15	edu.rit.m2mp.security.keyagree.TreeAgreement
33	11	edu.rit.m2mp.security.keyagree.Node
11	3	edu.rit.m2mp.security.keyagree.KeyProxy
334	38	edu.rit.m2mp.security.keyagree.Tree
9	3	edu.rit.m2mp.security.keyagree.MemberNode

```
12      6       edu.rit.m2mp.security.messages.AdminMessage
11      5       edu.rit.m2mp.security.messages.ApplicationMessage
10      9       edu.rit.m2mp.security.messages.GroupMessage
53      18      edu.rit.m2mp.security.messages.GroupMessageImpl
76      16      edu.rit.m2mp.security.messages.JoinGrant
84      16      edu.rit.m2mp.security.messages.JoinRequest
37      9       edu.rit.m2mp.security.messages.AgreeSet
7       3       edu.rit.m2mp.security.messages.ACK
18      4       edu.rit.m2mp.security.messages.AgreeAck
7       3       edu.rit.m2mp.security.messages.LeaveRequest
61      5       edu.rit.m2mp.security.messages.MessageConstants
125     4       edu.rit.m2mp.security.messages.MessageUtils
7       3       edu.rit.m2mp.security.messages.NACK
7       3       edu.rit.m2mp.security.messages.RefreshKeyRequest
7       3       edu.rit.m2mp.security.messages.NewKey
2       1       edu.rit.m2mp.security.signatures.hors.interfaces.HORSKey
3       2       edu.rit.m2mp.security.signatures.hors.interfaces.HORSKeyPairGenerator
5       4       edu.rit.m2mp.security.signatures.hors.interfaces.HORSParams
2       1       edu.rit.m2mp.security.signatures.hors.interfaces.HORSPrivateKey
2       1       edu.rit.m2mp.security.signatures.hors.interfaces.HORSPublicKey
21      5       edu.rit.m2mp.security.signatures.hors.spec.HORSParameterSpec
22      6       edu.rit.m2mp.security.signatures.hors.spec.HORSPrivateKeySpec
22      6       edu.rit.m2mp.security.signatures.hors.spec.HORSPublicKeySpec
166     14      edu.rit.m2mp.security.signatures.hors.test.FactoryTest
325     6       edu.rit.m2mp.security.signatures.hors.test.HORSTest
232     17      edu.rit.m2mp.security.signatures.hors.test.ProviderTest
117     2       edu.rit.m2mp.security.signatures.hors.test.SignExample
61      14      edu.rit.m2mp.security.signatures.hors.AlgIdHORS
255     24      edu.rit.m2mp.security.signatures.hors.AlgorithmId
154     20      edu.rit.m2mp.security.signatures.hors.PKCS8Key
149     22      edu.rit.m2mp.security.signatures.hors.X509Key
75      5       edu.rit.m2mp.security.signatures.hors.HORSKeyFactory
121     11      edu.rit.m2mp.security.signatures.hors.HORSKeyPairGenerator
41      5       edu.rit.m2mp.security.signatures.hors.HORSParameterGenerator
56      12      edu.rit.m2mp.security.signatures.hors.HORSParameters
58      8       edu.rit.m2mp.security.signatures.hors.HORSPrivateKey
15      1       edu.rit.m2mp.security.signatures.hors.HORSProvider
53      8       edu.rit.m2mp.security.signatures.hors.HORSPublicKey
145     14      edu.rit.m2mp.security.signatures.hors.HORSSignature
```

```
149     11      edu.rit.m2mp.security.signatures.SigningEngine
3       2       edu.rit.m2mp.security.signatures.SigningEngineDef
18      4       edu.rit.m2mp.security.utils.Arrays
37      7       edu.rit.m2mp.security.utils.Numbers
58      9       edu.rit.m2mp.security.utils.ThreadPool
20      5       edu.rit.m2mp.security.DefaultExceptionListener
3       2       edu.rit.m2mp.security.ExceptionHandler
3       2       edu.rit.m2mp.security.ExceptionListener
504     22      edu.rit.m2mp.security.MessageHandler
151     12      edu.rit.m2mp.security.ProtocolConstants
340     60      edu.rit.m2mp.security.Repository
123     31      edu.rit.m2mp.security.SecureChannel
4       3       edu.rit.m2mp.security.NullOutputListener
4       3       edu.rit.m2mp.security.NullExceptionListener
77      5       edu.rit.m2mp.security.ProviderInfo
3       2       edu.rit.m2mp.security.OutputListener
2       1       edu.rit.m2mp.security.OutputHandler
9       3       edu.rit.m2mp.security.DefaultOutputListener
324     3       edu.rit.m2mp.security.ChannelDemo

Average Object NCSS:                 92.45
Average Object Functions:            10.44
Average Object Inner Classes:         0.42
Average Object Javadoc Comments:      9.29
Program NCSS:                     8,662.00
```

Table D.2: Codebase Statistics – Lines of Code per Class

Bibliography

[1] *WikiPedia - Birthday Attack*. http://wikipedia.org, November 2003. See the *Birthday Attack* entry. Available online at http://en2.wikipedia.org/wiki/Birthday_attack. 47

[2] *WikiPedia - Elliptic Curve*. http://wikipedia.org, November 2003. See the *Elliptic Curve* entry. Available online at http://en2.wikipedia.org/wiki/Elliptic_curve. 46, 47

[3] *WikiPedia - Elliptic Curve Cryptography*. http://wikipedia.org, November 2003. See the *Elliptic Curve Cryptography* entry. Available online at http://en2.wikipedia.org/wiki/Elliptic_curve_cryptography. 47

[4] *WikiPedia - Integer Factorization*. http://wikipedia.org, January 2004. See the *Integer Factorization* entry. Available online at http://en.wikipedia.org/wiki/Integer_factorization. 44

[5] *WikiPedia - RSA*. http://wikipedia.org, February 2004. See the *RSA* entry. Available online at http://en.wikipedia.org/wiki/RSA. 44, 45

[6] Scott A Vanstone Alfred J. Menezes, Paul C.van Oorschot. *Handbook of Applied Cryptography*. CRC Press, 1996. 44

[7] American National Standards Institute (ANSI) for Financial Services. *X9.62-2000 – Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, November 2000. 44, 47

[8] American National Standards Institute (ANSI) for Financial Services. *X9.63-2001 – Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography*, November 2001. 47

[9] Giuseppe Ateniese, Michael Steiner, and Gene Tsudik. Authenticated group key agreement and friends. In *ACM Conference on Computer and Communications Security*, pages 17–26, 1998. Available online at http://citeseer.nj.nec.com/ateniese98authenticated.html. 50

[10] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. *Lecture Notes in Computer Science*, 1666:431–448, 1999. Available online at http://citeseer.nj.nec.com/bellare99forwardsecure.html. 24

[11] Ian Blake, Gadiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography*. Number 265 in London Mathematical Society Lecture Note Series. Cambridge University Press, 1999. 55, 56

[12] Jeremy Dahlgren. Efficient failure detection protocols for point-to-point communication networks. Master's thesis, Rochester Institute of Technology, 2004. 3

[13] RIT CS Department. Anhinga project. Department of Computer Science, Rochester Institute of Technology. Available from http://www.cs.rit.edu/~anhinga/. 3

[14] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976. 50

[15] FlexiProvider Research Group. Flexi core provider. Available online at http://flexiprovider.de. 27

[16] FlexiProvider Research Group. Flexi ec provider. Available online at http://flexiprovider.de. 27, 57

[17] Jorge Guajardo and Pedro Soria-Rodriguez. Implementation of the pohlig-hellman algorithm for the discrete logarithm problem, December 1995. Available online at http://alum.wpi.edu/ sorrodp/crypto/report.html. 47

[18] Vipul Gupta, Sumit Gupta, Sheueling Chang, and Douglas Stebila. Performance analysis of elliptic curve cryptography for ssl. In *Proceedings of the ACM workshop on Wireless security*, pages 87–94. ACM Press, 2002. Available online at http://doi.acm.org/10.1145/570681.570691. 47, 49

[19] Kieran S. Hagzan. Scalable and secure ad-hoc group communication. Master's thesis, Rochester Institute of Technology, 2004. 51, 73

[20] A. Jungmaier, E. Rescorla, and M. Tuexen. Transport layer security over stream control transmission protocol. Available online at http://www.ietf.org/rfc/rfc3436.txt. 74

[21] C. Jutla. Parallelizable encryption mode with almost free message integrity, 2000. Available online at http://citeseer.ist.psu.edu/jutla00parallelizable.html". 14

[22] Charanjit S. Jutla. Encryption modes with almost free message integrity. *Lecture Notes in Computer Science*, 2045:529–??, 2001. Available online at http://citeseer.ist.psu.edu/jutla00encryption.html. 14

[23] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Tree-based group key agreement. Technical report, Department of Information and Computer Science, University of California at Irvine, CA, USA, 2002. 9

[24] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Tree-based group key agreement. Technical Report 2002/009, Department of Information and Computer Science, University of California at Irvine, CA, USA, 2002. Available online at http://citeseer.nj.nec.com/kim02treebased.html. 9, 21, 31, 39

[25] RSA Laboratories. Cryptography faq – what are message authentication codes? Available online at http://www.rsasecurity.com/rsalabs/faq/2-1-7.html. 24

[26] A. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. Available online at http://citeseer.ist.psu.edu/lenstra99selecting.html. 49, 57

[27] Sun Microsystems. Java cryptography architecture. Available online at http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html. 58

[28] Sun Microsystems. Java reliable multicast service. Available from http://www.experimentalstuff.com/Technologies/JRMS/. 73

[29] Sun Microsystems. Sun community source license – java 2 platform, standard edition (J2SE). Available from http://www.sun.com/software/communitysource/j2se/java2/download.html. 107

[30] Roberto Di Pietro, Luigi V. Mancini, and Sushil Jajodia. Efficient and secure keys management for wireless mobile communications. In *Proceedings from the Second ACM International Workshop on Principles of Mobile Computing*, pages 66–73, 2002. 9, 11, 12

[31] Leonid Reyzin and Natan Reyzin. Better than biba: Short one-time signatures with fast signing and verifying. Technical report, Boston University, April 2002. 19, 24, 28, 33, 34

[32] Ohad Rodeh, Kenneth P. Birman, Mark Hayden, Zhen Xiao, and Danny Dolev. The architecture and performance of security protocols in the ensemble group communication system. Technical Report TR98-1703, Cornell and Hebrew Universities, 1998. 13

[33] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: a block-cipher mode of operation for efficient authenticated encryption. In *ACM Conference on Computer and Communications Security*, pages 196–205, 2001. 14

[34] J. P. Hubaux S. Capkun, L. Buttyan. Self-organized public-key management for mobile ad hoc networks. *IEEE Transactions on Mobile Computing*, 2(1):17, January - March 2003. Available online at `http://icawww.epfl.ch/Publications/Capkun/CapkunBH03tmc.pdf`. 71

[35] Michael Steiner, Gene Tsudik, and Michael Waidner. CLIQUES: A new approach to group key agreement. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 380–387, Amsterdam, 1998. IEEE Computer Society Press. Available online at `http://citeseer.nj.nec.com/steiner98cliques.html`. 9, 10, 11

[36] R. Stewart and Q. Xie et. al. Stream control transmission protocol. Available online at `http://www.ietf.org/rfc/rfc2960.txt`. 74

[37] J. Stone and R. Stewart et. al. Stream control transmission protocol checksum change. Available online at `http://www.ietf.org/rfc/rfc3309.txt`. 74

[38] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, 2002. 48

[39] Eric W. Weisstein. *CRC Concise Encyclopedia of Mathematics*. CRC Press, 2nd edition, December 2002. See the *Pollard Rho Factorization Method* section. Available online at `http://mathworld.wolfram.com/PollardRhoFactorizationMethod.html`. 44

[40] Eric W. Weisstein. *CRC Concise Encyclopedia of Mathematics*. CRC Press, 2nd edition, December 2002. See the *Field Characteristic* section. Available online at `http://mathworld.wolfram.com/FieldCharacteristic.html`. 45, 47

[41] Eric W. Weisstein. *CRC Concise Encyclopedia of Mathematics*. CRC Press, 2nd edition, December 2002. See the *Elliptic Curves* section. Available online at `http://mathworld.wolfram.com/EllipticCurve.html`. 46, 47