5-19-1999

Interactive and zero-knowledge proofs

Molli Noland

Rochester Institute of Technology
Mathematics and Statistics Department

Interactive and Zero-Knowledge Proofs

Molli Noland

A thesis submitted to
The Faculty of the Department of Mathematics and Statistics,
in partial fulfillment of the requirements for the degree of
Master of Science in Industrial and Applied Mathematics.

Approved by:

Stanisław P. Radziszowski

Theodore W. Wilcox

David S. Hart

George E. Christopher

May 19, 1999

WALLACE LIBRARY

Rochester Institute of Technology

RIT Thesis Binding Procedure
Examples of Thesis Reproduction Permission Statement

(3 options--please choose one)

Permission granted

Title of thesis Interactive and Zero-Knowledge Proofs

I, *author's name,* hereby **grant permission** to the Wallace Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: 5/19/99 Signature of Author: _____

OR

Permission From Author Required

Title of thesis _____

I, *author's name,* prefer to be contacted each time a request for reproduction is made. If permission is granted, any reproduction will not be for commercial use or profit. I can be reached at the following address:

_____

_____

_____

Phone: _____

Date: _____ Signature of Author: _____

OR

Permission Denied

Title of thesis _____

Abstract

An interactive proof involves two parties, the prover and the verifier. The goal of the proof is for the prover to convince the verifier that some instance of a decision problem is true. A zero-knowledge proof is an interactive proof where the only information learned by the verifier of the proof is the outcome of the proof. This thesis contains a theoretical overview of interactive and zero-knowledge proofs and describes experiments with implementations of some of them.

Two examples of interactive proofs from number theory are given, a protocol for quadratic non-residues and a protocol for subgroup non-membership. The third example of an interactive proof is a protocol for determining the truth value of a quantified Boolean formula. This interactive proof was implemented and the details of that implementation, plus a test of the implementation derived from game theory, are included. There is also a discussion of quantum interactive proofs. The two examples of perfect zero-knowledge proofs that are included are protocols for quadratic residues and for subgroup membership. These protocols were also implemented, and those details are included.

For each protocol, there is a discussion of the complexity status of the problems addressed by the protocol. There is also a brief discussion of the history and applications of interactive and zero-knowledge proofs.

Contents

List of Figures

Chapter 1

Complexity Theory

Before the definitions of interactive and zero-knowledge proofs are introduced, the reader should be familiar with the complexity status of the problems addressed by the protocols. In order to discuss the complexity status of these problems, it is necessary for the reader to understand a few key concepts of computational complexity theory.

1.1 Definitions

The complexity of decidable problems can be thought of in two ways: in terms of the time the problem requires or in terms of the space, or memory, that the problem requires. Both time and space complexity are measured in terms of n, the size of the input to the problem. First, we will discuss time complexity.

P is the class of all languages that are decidable in polynomial time on a deterministic Turing Machine; in other words, it is the set of languages that are decidable by $O(n^k)$ time Turing Machines, where k is any non-negative integer [Sipser, 235]. P represents the set of problems that can be realistically solved using a computer.

NP is the class of all languages that are decidable in polynomial time on a non-deterministic Turing Machine [Sipser, 244]. NP can also be thought of as the class of languages that can be verified by a polynomial time deterministic Turing Machine; in other words, the languages for which the membership relation can be realistically verified using a computer [Sipser, 243].

It is clear that P $\subseteq$ NP, since any language that can be decided by a polynomial time deterministic Turing Machine can clearly be verified by such a Turing Machine. It has been conjectured that P $\neq$ NP, but this has not been proven. NP is thought to contain languages that are not in P because an enormous amount of effort has been spent trying to find polynomial time algorithms for specific problems in NP without any success [Sipser, 247].

A language $\mathcal{L}$ is NP complete if $\mathcal{L} \in$ NP, and for every $\mathcal{L}' \in$ NP, there exists a polynomial transformation from $\mathcal{L}$ to $\mathcal{L}'$ [Sipser, 253]. A polynomial

transformation from $\mathcal{L}_1 \subseteq \Sigma_1^*$ to $\mathcal{L}_2 \subseteq \Sigma_2^*$ is a function $f : \Sigma_1^* \to \Sigma_2^*$ such that there is a polynomial time deterministic Turing Machine which computes f, and

$$\forall x \in \Sigma_1^* \; x \in \mathcal{L}_1 \text{ if and only if } f(x) \in \mathcal{L}_2 \text{ [Garey, 34].}$$

The idea of NP completeness is important because if P $\neq$ NP, then there can not exist a polynomial time algorithm for deciding any NP complete language. Also, if a polynomial time algorithm is found for any NP complete language, then the hierarchy collapses, and P = NP [Sipser, 248].

Now, we will move to the idea of space complexity. PSPACE is defined as the set of all languages that are decidable in polynomial space on deterministic Turing Machines [Garey, 170]. PSPACE is the space complexity analog of P. The analog of NP, NPSPACE, is defined in a similar way. It turns out that NPSPACE = PSPACE, since any non-deterministic Turing Machine, which uses $f(n)$ space, can be converted to a deterministic Turing Machine, which uses only $f^2(n)$ of space [Sipser, 279]. This is clearly still polynomial space.

P $\subseteq$ PSPACE since an polynomial-time machine can not use more than polynomial space [Sipser, 282]. For the same reason NP $\subseteq$ NPSPACE. The relationships between the classes of languages are then as follows:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE},$$

and both of the inclusions are conjectured to be strict.

PSPACE completeness is defined in the same way as NP completeness. A language $\mathcal{L}$ is PSPACE complete if for all $\mathcal{L}' \in$ PSPACE, there exists a polynomial transformation from $\mathcal{L}$ to $\mathcal{L}'$ [Garey, 171].

1.2 Complexity Status of Problems Addressed by Protocols

Now that the necessary concepts of complexity theory have been introduced, we can discuss the status of the problems addressed by the protocols used in this thesis. One of the fundamental assumptions made by the protocols is that the problem of factoring large numbers is hard. There is no known polynomial time algorithm for factoring large numbers, even though there has been extensive research in this area [Sipser, 339]. The complexity is measured in terms of $\log_2 n$, since this is the number of digits needed to represent the integer n. Many of the current cryptographic protocols, including RSA, are based on this assumption.

The problem of quadratic residues is posed as follows: given n and a, does there exist an x such that $x^2 \equiv a \bmod n$? If x exists, then a is a quadratic residue modulo n. This problem is solvable in polynomial time if n is prime, or if the factorization of n is known. But, if n is composite and the factorization is unknown, then the problem of quadratic residues is not known to be in P [Garey, 249].

The problem of subgroup membership is posed as follows: given n, a, and b, does there exist a k such that $b \equiv a^k \bmod n$? If k exists, then b is an element of the subgroup of $\mathcal{Z}_n^*$ generated by a. Finding k is equivalent to computing the discrete logarithm:

$$\log_a b \bmod n.$$

There is no known polynomial time algorithm for computing discrete logarithms, and if there was such an algorithm, then we would have the ability to factor numbers in polynomial time [Schneier, 262].

The last problem that the protocols address is determining the truth value of a quantified Boolean formula (QBF). The problem is posed as follows: given a set of Boolean variables $\{x_1, x_2, \ldots, x_n\}$ and a well-formed QBF

$$B = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \; E(x_1, x_2, \ldots, x_n),$$

where $Q_i \in \{\forall, \exists\}$ and E is a quantifier-free Boolean expression, is B true? This problem is PSPACE complete, and a proof of this can be found in [Sipser, 284-287]. If the QBF is in conjunctive normal form with no more than two literals per clause, then the problem is solvable in polynomial time [Garey, 262].

Chapter 2

Interactive Proofs

Let P be a decision problem, then an interactive proof for P involves two parties, the prover (Peggy) and the verifier (Vic). Peggy is trying to convince Vic that she knows whether or not the given $x \in P$. At the end of the proof Vic should be convinced that Peggy is either telling the truth or she is not. This process is made up of rounds. Each round consists of a challenge by Vic and a response by Peggy. In each round, Peggy and Vic alternately do the following: receive a message, do a computation, and send a message. At the end of the process, Vic either accepts or rejects Peggy's proof. For the proofs that we wil consider, Peggy has unlimited computational power, but Vic is limited to polynomial time algorithms.

An interactive proof must meet two conditions: completeness and soundness. For an interactive proof to be complete, Vic will always accept Peggy's proof if x is a yes-instance of the decision problem. The proof will be sound if there is only a small probability that Vic will accept Peggy's proof if x is a no-instance of the decision problem. In other words, the interactive protocol becomes an interactive proof if the probability of Type I error, α, is zero, and the probability of Type II error, β, is small. β can be reduced arbitrarily close to zero by iterating the process a certain number of times.

2.1 Number Theoretical Examples

2.1.1 Quadratic Non-Residues

A quadratic non-residue is a number a, such that the equation

$$x^2 \equiv a \bmod n$$

has no solutions for $x \in \mathcal{Z}_n^*$. Currently, there does not exist a polynomial time algorithm which decides if a is a quadratic non-residue modulo n; therefore, Vic would not be able to decide this problem on his own. He can decide the problem with the help of Peggy by using an interactive proof. Figure 2.1 describes an interactive proof for quadratic non-residues.

Input: An integer n with unknown factorization $n = pq$, where p and q are prime, and $x \in \mathcal{Z}_n^*$, where x is a quadratic non-residue mod$n$.

1. Repeat the following steps $\log_2 n$ times:
   (a) Vic chooses a random $v \in \mathcal{Z}_n^*$ and computes $y = v^2 \bmod n$. Vic chooses $i \in \{0, 1\}$ at random. He sends $z = x^i y \bmod n$ to Peggy.
   (b) If z is a quadratic residue modulo n, then Peggy defines $j = 0$; otherwise, she defines $j = 1$. Peggy sends j to Vic.
   (c) Vic checks to make sure that $i = j$
2. Vic accepts Peggy's proof, that x is a quadratic non-residue mod n, if step c is verified in each of the $\log_2 n$ rounds.

Figure 2.1: Interactive Proof for Quadratic Non-Residues [Stinson, 408]

To prove that this proof is an interactive proof, it needs to be shown that the proof is complete and sound. The proof is complete if for every x that is a quadratic non-residue, Vic will always accept Peggy's proof.

Theorem 2.1 *The protocol in Figure 2.1 provides a complete proof for quadratic non-residues.*

> **Proof:** Assume that x is a quadratic non-residue and that Vic rejects the proof. If Vic rejects the proof, then $i \neq j$ in at least one of the $\log_2 n$ rounds. If $i \neq j$, then either
>
> 1. z is a quadratic residue and $z = xv^2 \bmod n$, or
> 2. z is quadratic non-residue and $z = v^2 \bmod n$.
>
> The second case is an obvious contradiction by the definition of quadratic non-residue, so this leaves only the first case to prove. If z is a quadratic residue, then there exists a $w \in \mathcal{Z}_n^*$, such that $w^2 \equiv z \bmod n$; therefore,
>
> $$z \equiv xv^2 \bmod n \text{ and } w^2 \equiv z \bmod n$$
> $$w^2 \equiv xv^2 \bmod n$$
> $$(wv^{-1})^2 \equiv x \bmod n.$$
>
> This is a contradiction since x is a quadratic non-residue modulo n. Note that v^{-1} exists since $v \in \mathcal{Z}_n^*$. Since both cases caused contradictions, the assumption must be false: If x is a quadratic non-residue, then Vic will always accept Peggy's proof; therefore, the proof is complete.

Soundness is proved by showing that if x is a quadratic residue modulo n, then the probability that Vic accepts Peggy's proof is very small.

Theorem 2.2 *The protocol in Figure 2.1 provides a sound proof for quadratic non-residues.*

> **Proof:** x is a quadratic residue modulo n; therefore, there exists a $w \in \mathcal{Z}_n^*$, such that $w^2 \equiv x \bmod n$. Peggy will always send $j = 0$ in step b since if $i = 0$, then $z = v^2 \bmod n$, and if $i = 1$, then
>
> $$z = xv^2 \bmod n \text{ and } w^2 \equiv x \bmod n$$
> $$z \equiv (wv)^2 \bmod n;$$
>
> and therefore, z is a quadratic residue in both cases. Since j always equals zero, the probability of i being equal to j is the same as the probability of i being zero. Since i is chosen at random from $\{0, 1\}$, the probability of i being zero is one-half. This procedure is repeated $\log_2 n$ times; therefore, the probability of Vic accepting Peggy's proof if x is a quadratic residue is
>
> $$\frac{1}{2^{\log_2 n}} = 1/n.$$
>
> This means that for large n the protocol for quadratic non-residues is sound.

By Theorem 2.1 and Theorem 2.2, the protocol for quadratic non-residues, as presented in Figure 2.1, is an interactive proof since it is both complete and sound.

2.1.2 Subgroup Non-Membership

The second example of interactive proofs asks the question, is b a member of the subset of $\mathcal{Z}_n^*$ generated by a? In other words, does there exist a $k \in \{0, 1, \ldots, m-1\}$, such that $b \equiv a^k \bmod n$, if $a, b \in \mathcal{Z}_n^*$ and a has order m in $\mathcal{Z}_n^*$? As with the problem of quadratic non-residues, there is no known polynomial time algorithm for deciding if there exists such a k, so Vic needs to use an interactive proof to decide this problem. The proof in Figure 2.2 is an interactive proof for subgroup non-membership, the case when the answer to the above question is no.

Note that even though n is large, a^k can be computed efficently by using binary exponentiation. Using the square-and-multiply algorithm, as presented in [Stinson, 127], this calculation can be done in polynomial time. Also notice that m should be a number less than $\phi(n)$, since

$$\forall a \in \mathcal{Z}_n^* \; a^{\phi(n)} \equiv 1 \bmod n.$$

If $m = \phi(n)$ then the subgroup generated by a is $\mathcal{Z}_n^*$, and b will always be a subgroup member.

Theorem 2.3 and Theorem 2.4 state the completeness and soundness of this proof, and therefore, verify that the proof is an interactive proof.

Input: A positive integer n and two distinct elements $a, b \in \mathcal{Z}_n^*$, where the order of a is denoted by m and is publicly known.

1. Repeat the following steps $\log_2 n$ times:

   (a) Vic chooses $j \in \{0, 1, \ldots, m-1\}$ and $i \in \{0, 1\}$ at random. He then computes
   $$g = a^j b^i \bmod n$$
   and sends g to Peggy.

   (b) If $g \equiv a^k \bmod n$, where $k \in \{0, 1, \ldots, m-1\}$, then Peggy defines $t = 0$; otherwise, she defines $t = 1$. Peggy sends t to Vic.

   (c) Vic checks to make sure that $i = t$.

2. Vic accepts Peggy's proof, that b is not a member of the subgroup of $\mathcal{Z}_n^*$ generated by a, if step c is verified in each of the $\log_2 n$ rounds.

Figure 2.2: Interactive Proof for Subgroup Non-Membership

Theorem 2.3 *The protocol in Figure 2.2 provides a complete proof for subgroup non-membership.*

Proof: Show that if b is not a member of the subgroup of $\mathcal{Z}_n^*$ generated by a, then Vic will always accept Peggy's proof for subgroup non-membership. Assume that b is not a member, and that Vic rejects Peggy's proof. Vic will only reject the proof if for at least one of the $\log_2 n$ rounds $i \neq t$. This means that either

1. $g = a^j \bmod n$ and $g \not\equiv a^k \bmod n$, for all $k \in 0, 1, \ldots, m-1$, or
2. $g = a^j b \bmod n$ and $g \equiv a^k \bmod n$, for some $k \in 0, 1, \ldots, m-1$.

The first case is clearly a contradiction. This leaves the second case:

$$\begin{aligned} g = a^j b \bmod n \text{ and } g &\equiv a^k \bmod n \\ a^k &\equiv a^j b \bmod n \\ b &\equiv a^{k-j \bmod m} \bmod n \end{aligned}$$

This is also a contradiction, since we assumed that b was not a member of the subgroup generated by a. Since both cases lead to contradictions, the initial assumption is wrong; therefore, if b is not a member of the subgroup generated by a then Vic will always accept Peggy's proof. This proves the completeness of the protocol for subgroup non-membership.

Theorem 2.4 *The protocol in Figure 2.2 provides a sound proof for subgroup non-membership.*

Proof: Show that if b is a member of the subgroup of $\mathcal{Z}_n^*$ generated by a, then there is only a small probability that Vic will accept Peggy's proof for subgroup non-membership. Vic only accepts the proof if $i = t$ for all $\log_2 n$ rounds. Since b is a subgroup member, $b \equiv a^k \bmod n$, for some $k \in \{0, 1, \ldots, m-1\}$. Peggy will always send $t = 0$ in step b, since either $g = a^j \bmod n$ or

$$g = a^j b \bmod n \text{ and } b \equiv a^k \bmod n$$
$$g \equiv a^{j+k \bmod m} \bmod n;$$

and therefore, g is a subgroup member. Since t always equals zero, i will always equal t whenever i equals zero. The probability of i being zero is one-half, since i is chosen at random from {0,1}. This procedure is repeated $\log_2 n$ times; therefore, as seen before, the probability of Vic accepting Peggy's proof, if b is a member of the subgroup generated by a, is $1/n$. This means the protocol for subgroup non-membership is sound.

2.2 Quantified Boolean Formulas

A third example of an interactive proof is a protocol for determining the truth value of a quantified Boolean formula. Some definitions and theorems are needed before the proof can be presented. The information in Section 2.2.1 and Section 2.2.2 can be found in more detailed form in [Shamir].

2.2.1 Preliminary Information

The class of quantified Boolean formulas (QBF) is defined as the closure of the set of Boolean variables (x_i) and their negations ($\overline{x_i}$) under the operations $\wedge$ (and), $\vee$ (or), $\forall x_i$ (universal quantification), and $\exists x_i$ (existential quantification). It is not required that all quantifiers appear in a leftmost prefix [Shamir, 870].

A QBF is considered **closed** if all its variables are quantified. The truth value of a closed QBF can be evaluated. A convenient measure of the **size** of a QBF is the number of distinct variables that the QBF contains. A closed QBF is **simple** if every occurrence of each variable is separated from its point of quantification by at most one universal quantifier [Shamir, 870]. By this definition,

$$\forall x_1 \exists x_2 [(x_1 \vee \overline{x_2}) \vee \forall x_3 (x_1 \wedge x_3)]$$

is simple. The first instance of x_1 is separated from its point of quantification by no universal quantifiers, and the second instance is separated only by $\forall x_3$. The instances of x_2 and x_3 are separated from their points of quantification by no universal quantifiers. On the other hand,

$$\forall x_1 \forall x_2 \forall x_3 [(x_1 \wedge x_2) \vee x_3]$$

is not simple since x_1 is separated from its point of quantification by both $\forall x_2$ and $\forall x_3$. Even though this definition seems to restrict the power of QBF's, it can be proved that:

Theorem 2.5 *Every QBF of size n can be transformed into an equivalent simple QBF whose size is polynomial in n [Shamir, 870].*

Closed QBF's can be transformed into arithmetic expressions by the following process, which Shamir calls **arithmetization** [Shamir, 871]:

1. Replace each occurrence of the Boolean variable x_i by a variable $z_i \in \mathcal{Z}$.
2. Replace each occurrence of $\overline{x_i}$ by $(1 - z_i)$.
3. Replace $\wedge$ (and) by (integer multiplication), $\vee$ (or) by $+$ (integer addition), $\forall x_i$ (universal quantification) by $\prod_{z_i \in \{0,1\}}$ (integer product), and $\exists x_i$ (existential quantification) by $\sum_{z_i \in \{0,1\}}$ (integer sum).

According to this process, the arithmetization of

$$\forall x_1 \exists x_2 [(x_1 \vee \overline{x_2}) \vee \forall x_3 (x_1 \wedge x_3)]$$

would be

$$\prod_{z_1 \in \{0,1\}} \sum_{z_2 \in \{0,1\}} [(z_1 + (1 - z_2)) + \prod_{z_3 \in \{0,1\}} (z_1 \cdot z_3)].$$

From this definition of the process of arithmetization, Theorem 2.6 can easily be proved by induction on the structure of QBF's:

Theorem 2.6 *A closed QBF B is true if and only if the value of its arithmetic form A is non-zero [Shamir, 871].*

Theorem 2.6 can be proved by induction on the structure of B.

One problem with using the arithmetic form to find the truth value of the QBF is that if the QBF is true, the value of A can be very large, but Theorem 2.7 states that there exists an upper-bound to the value of A:

Theorem 2.7 *Let B be a closed QBF of size n. Then the value of its arithmetic form A can not exceed $\mathcal{O}(2^{n^n})$ [Shamir, 871].*

Using Theorem 2.6 and Theorem 2.7, it can be proved that the following equivalence is true, and therefore, the equivalence can be used to find the truth value of a QBF:

Theorem 2.8 *Let B be a closed QBF of size n. Then there exists a prime p of length polynomial in n such that $A \not\equiv 0 \bmod p$ if and if B is true [Shamir, 872].*

Definitions of the **functional form** and **randomized form** of a QBF are also needed. Given a closed QBF B, the **functional form** A' of its arithmetic expression A is found by eliminating the leftmost $\prod_{z_i \in \{0,1\}}$ or $\sum_{z_i \in \{0,1\}}$. A' is

then a polynomial function of one free variable, and this polynomial function will be denoted $q(z_i)$ [Shamir, 872]. By this definition, the functional form of the QBF

$$B = \forall x_1 \exists x_2 [(x_1 \vee \overline{x_2}) \vee \forall x_3 (x_1 \wedge x_3)]$$

is

$$A' = \sum_{z_2 \in \{0,1\}} [(z_1 + (1 - z_2)) + \prod_{z_3 \in \{0,1\}} (z_1 \cdot z_3)],$$

and the polynomial description of A' is

$$\begin{aligned} q(z_1) &= ((z_1 + (1-0)) + (z_1 \cdot 0)(z_1 \cdot 1)) + ((z_1 + (1-1)) + (z_1 \cdot 0)(z_1 \quad 1)) \\ &= ((z_1 + 1) + 0) + ((z_1 + 0) + 0) \\ &= 2z_1 + 1. \end{aligned}$$

The **randomized form** of A is simply $A'(z_i = r)$, where r is a random element of $\mathcal{Z}_p$ [Shamir, 872]. The value of $A'(z_i = r)$ is $q(r)$.

During the interactive proof, Peggy will need to split A into A_1 and A_2. This is done by letting A_1 be everything in A before the first $\prod_{z_i \in \{0,1\}}$ or $\sum_{z_i \in \{0,1\}}$, except the rightmost $+$ or $\cdot$, and letting A_2 be everything after, and including, the first $\prod_{z_i \in \{0,1\}}$ or $\sum_{z_i \in \{0,1\}}$ [Shamir, 874]. For example, if Peggy is splitting

$$A = 1 + \sum_{z_2 \in \{0,1\}} (1 \cdot z_2),$$

then

$$A = A_1 + A_2,\ A_1 = 1,\ \text{and}\ A_2 = \sum_{z_2 \in \{0,1\}} (1 \cdot z_2),$$

and if Peggy is splitting

$$A = \prod_{z_1 \in \{0,1\}} \sum_{z_2 \in \{0,1\}} (z_1 + z_2),$$

then $A_2 = A$ and A_1 is empty.

2.2.2 Determining the Truth Value of a QBF

The interactive proof presented in Figure 2.3 takes a closed, simple QBF as input and proves that the value of A, the arithmetic form of B, is $a \bmod p$, where a is a non-negative integer and p is a prime that is polynomially long in the size of B. If Peggy claims that a is zero, then the proof is determining the truth value of the QBF B; if the proof accepts Peggy's claim then B is false, otherwise B is true.

Since Vic is restricted to polynomial time algorithms, he is not capable of deriving the polynomials $q(z_i)$ from A, and this is why Peggy is necessary in the proof. The QBF B must be simple so that Vic is able to evaluate the polynomials $q(z_i)$ in polynomial time. If B was not required to be simple, Vic

Input: A closed, simple QBF B and the arithmetic form A.

1. Either Vic or Peggy choose a prime p that is polynomially long in the size of B [Shamir, 872].
2. Peggy sends the claimed value a, of $A \bmod p$, to Vic.
3. While the algorithm is not done, repeat the following:

   Vic splits A into $A_1 \cdot A_2$ or $A_1 + A_2$.

   IF A_2 is empty, the algorithm is finished and Vic accepts Peggy's claim if and only if

   $$a \equiv a_1 \bmod p,$$

   where a_1 is the value of A_1.

   ELSE IF A_1 is non-empty, Vic sets

   $$A = A_2 \bmod p \text{ and } a = a/a_1 \bmod p \text{ or } a = a - a_1 \bmod p,$$

   depending on if A is $A_1 \cdot A_2$ or $A_1 + A_2$. If

   $$A = A_1 \cdot A_2 \text{ and } a_1 \equiv 0 \bmod p,$$

   then the algorithm is finished and Vic accepts Peggy's claim if and only if

   $$a \equiv 0 \bmod p.$$

   ELSE Vic needs more information from Peggy.

   (a) Peggy sends the polynomial description, $q(z_i)$ of A' to Vic.

   (b) Vic checks that

   $$a \equiv q(0) \cdot q(1) \bmod p \text{ or } a \equiv q(0) + q(1) \bmod p,$$

   depending on the first symbol of A_2. If this fails, the algorithm is finished, and Vic rejects Peggy's claim; otherwise, Vic sends a random $r \in \mathcal{Z}_p$ to Peggy and sets

   $$A = A'(z_i = r) \bmod p \text{ and } a = q(r) \bmod p.$$

Figure 2.3: Interactive Proof for Determining the Truth Value of a Closed, Simple QBF [Shamir, 874]

would not necessarily be able to evaluate the polynomial $q(z_i)$. For simple QBF's Vic is guarenteed to be able to handle these polynomials because of the following theorem:

Theorem 2.9 *If B is simple, then the degree of the polynomial $q(z_i)$, that describes the functional form of A, grows at most linearly with the size of B [Shamir, 873].*

Before the protocol given in Figure 2.3 is proved to be an interactive proof, we will demonstrate a small numerical example of the protocol. If we let

$$B = \forall x_1 \exists x_2 (x_1 \wedge \overline{x_2}),$$

then A, the arithmetized version of B, is

$$\prod_{z_1 \in \{0,1\}} \sum_{z_2 \in \{0,1\}} [z_1 \cdot (1 - z_2)].$$

In step 1, set p to 17, and in step 2, Peggy sends the claimed value of a, which in this case will be 0. Now, step 3 is repeated until one of the ending conditions is met.

In the first iteration A_1 is empty and $A_2 = A$, meaning that neither the if nor the else if clause holds, and Vic must request more information from Peggy. Peggy sends Vic

$$q(z_1) = z_1.$$

Vic must check that $a \equiv q(0) \cdot q(1) \bmod p$, since a universal quantifier was removed to form A':

$$q(0) \cdot q(1) = 0 \cdot 1 = 0 \equiv a \bmod 17,$$

so this step checks. Vic then chooses a r at random. We will let $r = 5$. He then sets

$$\begin{aligned} A &= A'(z_1 = 5) = \sum_{z_2 \in \{0,1\}} [5 \cdot (1 - z_2)] \text{ and} \\ a &= q(5) \equiv 5 \bmod 17. \end{aligned}$$

Now, step 3 must be repeated. Again, A_1 is empty and $A_2 = A$, so Vic asks Peggy for additional information. Peggy sends Vic

$$q(z_2) = 5 - 5z_2.$$

Vic checks that $a \equiv q(0) + q(1) \bmod p$, since the removed quantifier was existential:

$$q(0) + q(1) = (5 - 5 \cdot 0) + (5 - 5 \cdot 1) = 5 \equiv a \bmod 17.$$

Since this step checked, Vic then chooses r, at random, to be 9, and sets

$$\begin{aligned} A &= A'(z_2 = 9) = [5 \cdot (1 - 9)] = -40 \equiv 11 \bmod 17 \text{ and} \\ a &= q(9) = (5 - 5 \cdot 9) = -40 \equiv 11 \bmod 17. \end{aligned}$$

Repeating step 3 a final time, $A_1 = 11$ and A_2 is empty. This meets the if condition, so Vic checks that $a \equiv a_1 \bmod p$:

$$a = 11 \equiv a_1 \bmod 17.$$

This check is verified; therefore, Vic accepts Peggy's proof that the QBF B is false.

Now that we have seen how the protocol works, it needs to be shown that the protocol is an interactive proof. This is done by Theorem 2.10, which proves completeness and soundness of the protocol.

Theorem 2.10 *If Peggy chooses the prime p, then the protocol given in Figure 2.3 provides a complete and sound proof for determining the truth value of a closed, simple QBF.*

> **Proof**: The proof is complete since Peggy will always be able to justify her claimed values and polynomials if she is telling the truth, and Vic must accept the proof if Peggy justifies her claims at every step [Shamir, 875]. Note that if Vic chooses the prime, the proof does not have perfect completeness since if p is a divisor of the true value of A, then B will appear to be false when in reality it is true [Shamir, 872].
>
> The proof is sound since if Peggy provides an incorrect value of a, then she is forced to provide an incorrect polynomial $q(z_i)$ to support her claim. An incorrect polynomial of degree t can agree with the correct polynomial on at most t of the p values in $\mathcal{Z}_p$. Since the value of p is exponential in the size of B and the value of r is chosen by Vic, the probability is negligible that the incorrect $q(z_i)$ will yield a correct value when q is evaluated at r. Peggy is therefore providing incorrect values for smaller and smaller subexpressions, so there is a exponentially small probability that Vic will not expose Peggy when he evaluates the final subexpression. [Shamir, 875]

2.2.3 Implementation of the Interactive Proof

`qbf` is a program which implements the algorithm described in Figure 2.3. The program should be thought of as Peggy, and the user should be thought of as Vic. The only required input to the program is an input file which contains a QBF. The QBF is assumed to be closed and simple, and it assumed to have proper syntax. Vic has the option of providing the value a of the QBF. If Vic does not choose a value for a, Peggy assumes it to be zero, making it a protocol for non-membership in QBF. Vic also has the option to provide the value of p. If p is chosen by Peggy, p will have one hundred digits.

The output of `qbf` is a transcript of the interactions between Peggy and Vic. The inputs, the messages exchanged, and the outcome of the proof are printed to the screen. Instances of QBF's which may lead to applications should have at least 100 variables and 400 clauses. Sample runs of the program are found

in Appendix B.1. The next section discusses the details of one of the formulas used to test `qbf`.

2.2.4 QBF and a Pebbling Problem

Since QBF is PSPACE complete, interesting (i.e. difficult) instances of QBF's can be found by transforming other PSPACE complete problems into QBF's. Games are particularly suited for this purpose. One type of test case for the program `qbf` was derived from the pebbling game. The pebbling game is played on an acyclic, directed graph, where every node has at most two predecessors. The player has a set of pebbles, and he can do one of the following at each time step:

1. He can remove a pebble from any vertex at any time.
2. He can place a pebble on any unpebbled vertex v if and only if all the predecessors of v are currently pebbled.
3. He can move a pebble to an unpebbled vertex v from a predecessor of v if and only if all predecessors of v are currently pebbled.

The object of the game is to pebble a specified goal vertex [Gilbert, 513]. The pebbling problem is defined as follows: "Given a graph G, can a vertex v in G be pebbled using no more than s pebbles?" [Gilbert, 514]. This problem is PSPACE complete, as proven in [Gilbert]; therefore, there is a transformation from QBF to the pebbling problem.

A natural transformation can be devised if the QBF is in conjunctive normal form, with exactly three literals per clause and all of the quantifiers in the prefix. This means that the QBF is of the following form:

$$Q_1 x_1 Q_2 x_2 \ldots Q_n x_n((l_{1,1} \vee l_{1,2} \vee l_{1,3}) \wedge (l_{2,1} \vee l_{2,2} \vee l_{2,3}) \wedge \ldots \wedge (l_{m,1} \vee l_{m,2} \vee l_{m,3})),$$

where $Q_i \in \{\forall, \exists\}$, x_i's are Boolean variables, $l_{j,k} \in \{x_i, \overline{x_i}\}$, n is the number of quantifiers/variables, and m is the number of clauses. From this QBF, a graph G can be constructed, with goal vertex q_1, such that the QBF is true if and only if q_1 can be pebbled with $s = 3n + 3$ pebbles [Gilbert, 514].

A pyramid of vertices is defined as in Figure 2.4, and it can be abbreviated as shown in the figure. It takes at least k pebbles to pebble a k-pyramid [Gilbert, 515]. G consists of $n + m$ blocks of vertices, one for each quantifier and one for each clause in the formula. The universal and existential quantifier blocks, q_i, are shown in Figure 2.5 and Figure 2.6, respectively. The block for each clause, p_j, is shown in Figure 2.7. p_0 is simply a single vertex, and $p_m = q_{n+1}$ [Gilbert, 516].

Using the preceeding definitions the graph G, derived from

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \exists x_5((\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_1 \vee x_4 \vee x_5) \wedge (x_3 \vee \overline{x_4} \vee \overline{x_5})),$$

is shown in Figure 2.8. The blocks for each of the five quantifiers are located on the left side of the graph, and the blocks for each clause are on the right side of

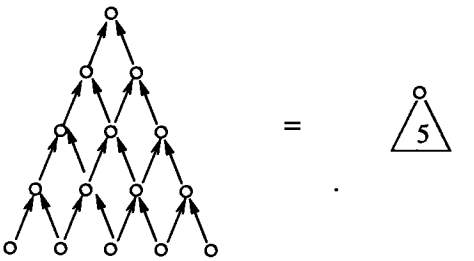

Figure 2.4: A 5-pyramid [Gilbert, 515]

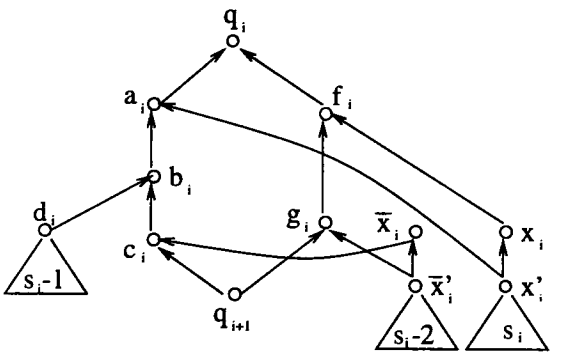

Figure 2.5: Universal Quantifier Block [Gilbert, 517]

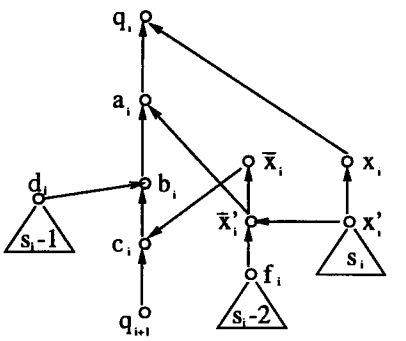

Figure 2.6: Existential Quantifier Block [Gilbert, 517]

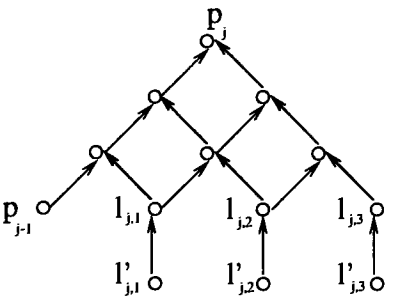

Figure 2.7: Clause Quantifier Block [Gilbert, 517]

the graph. The edges from the quantifier blocks to the clause blocks represent the occurrences of the variables in the clauses.

The proof that this process correctly produces a graph G such that the QBF is true if and only if q_1 can be pebbled with s pebbles can be found in [Gilbert]. This example was one test case used in testing `qbf`, and the output from this test can be found in Appendix B.1. `qbf` showed that the above QBF is true; therefore, q_i can be pebbled with eighteen pebbles.

Several other games, including Go and Hex, have been shown to be PSPACE complete. Refer to [Garey] for a discussion of some games that are PSPACE and NP complete. [Sipser] contains a detailed description of a transformation from the game Geography to QBF. This transformation is very similar to the one we have just described. Other examples of QBF's, created by Jussi Rintanen, can be found at [Rintanen].

2.2.5 IP = PSPACE

IP is the set of all languages that have efficient interactive proofs of membership [Shamir, 869]. Shamir has proved that IP and PSPACE are equivalent. Using the fact that QBF is PSPACE complete, we can prove the following theorem:

Theorem 2.11 *IP = PSPACE.*

> **Proof:** First, it needs to be shown that IP $\subseteq$ PSPACE. This is easy since every IP language is accepted by the PSPACE machine that simply traverses the tree of all possible interactions between Peggy and Vic [Shamir, 869]. This machine is a PSPACE machine since the depth of the tree is polynomial, and at each level only a polynomial amount of memory is needed.
>
> Second, it needs to be shown that PSPACE $\subseteq$ IP. This is done by using the IP for deciding QBF. The interactive proof presented here requires that the QBF be closed and simple, but as we mentioned in Section 2.2.1, any instance of QBF can be transformed into a

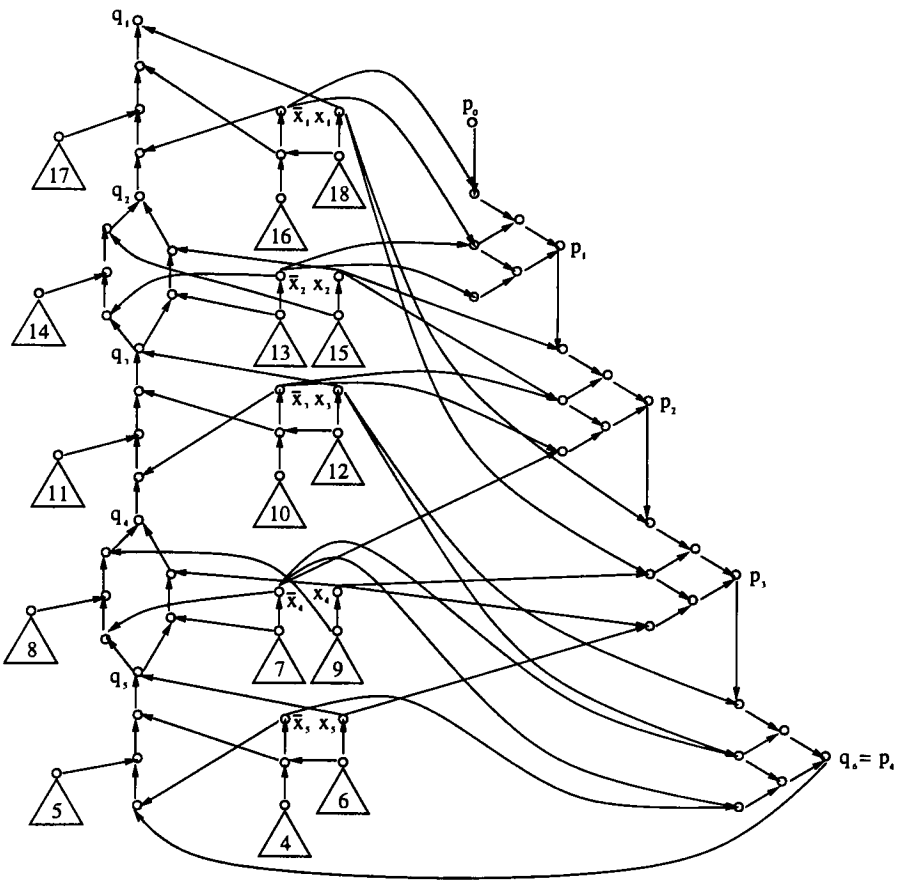

Figure 2.8: Graph for $\exists x_1 \forall x_2 \exists x_3 \forall x_4 \exists x_5 ((\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_1 \vee x_4 \vee x_5) \wedge (x_3 \vee \overline{x_4} \vee \overline{x_5}))$, where the maximum number of pebbles allowed is 18 [Gilbert, 518].

closed, simple QBF. Since there is an IP for every instance in QBF, and QBF is PSPACE complete, then PSPACE $\subseteq$ IP.

Since IP $\subseteq$ PSPACE and PSPACE $\subseteq$ IP, IP = PSPACE.

2.3 Quantum Interactive Proofs

Once quantum computers are built, the entire concept of tractability will likely change. Because of this change in the definition of what is realistically computable, only some interactive protocols will remain secure. The idea of interactive proof systems has been generalized to the idea of quantum computing by giving a definition of quantum interactive proof systems. This idea is discussed at length in [Watrous]. Also, an audio presentation of this paper and other quantum computing information can be found at [AQIP99]. In this thesis, quantum interactive proof systems will be defined, and a quantum interactive proof for the truth value of a QBF will be given. First some definitions are needed.

2.3.1 Preliminary Information

A quantum computer differs from a classical computer because the quantum computer can not only read and write zeros and ones, but it can read and write superpositions of zeros and ones simultaneously [Williams, 24]. A single bit of a quantum computer, a **qubit**, is thought of as a vector within a sphere which simultaneously measures the zero-ness and one-ness of the bit, as well as the phase [Williams, 25]. The values $|0\rangle$ and $|1\rangle$ are written to correspond to the bits 0 and 1. A quantum computer is built of quantum gates, which are the same as regular gates, except they input and output qubits rather than bits. Quantum circuits, which are made up of quantum gates, are reversible, and the transformation from the inputs to the outputs is thought of as a unitary operator [Williams, 44].

Quantum computers, when built, will have a dramatic impact on our concept of tractability. Peter Shor has developed efficent quantum algorithms for factoring large integers and solving discrete logarithms on quantum computers [Shor]. Since RSA and other cryptographic systems are based on the belief that there is no efficent algorithm for these problems, new secure protocols will need to be developed. This is also true for several interactive protocols, including the ones discussed in this thesis. New methods of quantum cryptography are being created to replace the methods that will become insecure once quantum computers are built. One such method was developed by Charles Bennett and Giles Brassard [Bennett]. This method encodes messages using polarized photons, and it obtains its security from the impossibility of cloning quantum information and from the consequences of Heisenberg's uncertainty principle [Williams, 164].

Quantum computers will also extend the realm of what is imaginable. Since randomness is at the heart of nature, and hence at the heart of quantum physics,

quantum computers will be able to generate true random numbers; whereas, classical computers can only pretend to generate random numbers [Williams, 147-148]. Quantum computers will also make teleportation feasible. This is done by exploiting the ideas of entanglement of states and non-logical influence [Williams, 184]. Teleportation would provide an ultra-secure method for sending data, since the data would never need to cross a channel, it would simply be teleported from one secure system to another [Williams, 185].

Now that we have seen the consequences of quantum computing, we are ready to define the concept of a quantum interactive proof. A k-round verifier Vic is thought of as a polynomial time computable mapping

$$V : \Sigma^* \times \{0, 1, \ldots, k\} \rightarrow \Sigma^*,$$

where $V(x, j)$ is an encoding of a quantum circuit. That circuit is made up of quantum gates which are chosen from a universal set of gates which necessarily includes the Walsh-Hadamard gate and a gate for reversible computation. Each circuit $V(x, j)$ must be polynomial in size. The circuit acts on two sets of qubits, the message and the ancilla. The message is the communication line between Vic and Peggy, and the ancilla is Vic's private information storage [Watrous, 3]. A k-round prover Peggy is a mapping P from $\Sigma^* \times \{0, 1, \ldots, k\}$ to the set of all quantum circuits. There are no restrictions on the complexity of $P(x, j)$, and $P(x, j)$ also acts on two sets of qubits, the common message and her ancilla [Watrous, 3].

An example of the pair (P, V) can be seen in Figure 2.9. The probability that (P, V) accepts x is defined as the probability that the output qubit yields $|1\rangle$ when the circuits are applied in sequence and all qubits are initially in the $|0\rangle$ state [Watrous, 3]. A language $\mathcal{L}$ has a k-round quantum interactive proof system, with error probability ϵ, if there exists a k-round verifier Vic such that:

1. There exists a k-round prover Peggy such that for all $x \in \mathcal{L}$, (P, V) accepts x with probability one.

2. For all $x \notin \mathcal{L}$, and for all k-round provers P', (P', V) accepts x with probability at most ϵ [Watrous, 3].

Before the quantum interactive proof for QBF is presented, another interactive proof for QBF needs to be discussed because this interactive proof is used in the quantum interactive proof for QBF.

2.3.2 Second Interactive Proof for QBF

The second interactive proof for determining the truth value of a QBF is due to Shen, and the proof can be found in its original form in [Shen]. This thesis will discuss a modification of Shen's proof from [Sipser, 364-366]. The proof requires that the QBF ϕ be of the form

$$\phi = Q_1 x_1 Q_2 x_2 \ldots Q_m x_m \; B(x_1, x_2, \ldots, x_m),$$

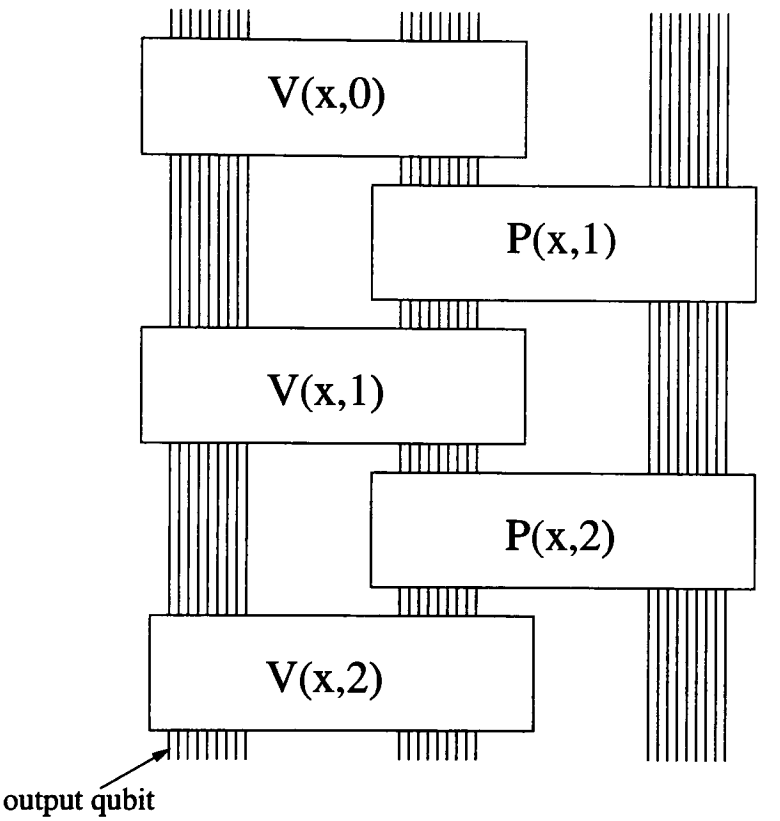

Figure 2.9: Quantum circuit for a two-round quantum interactive proof system [Watrous, 4].

where $Q_i \in \{\forall, \exists\}$, x_i's are Boolean variables, and B is a quantifier-free Boolean formula. b is the polynomial associated with B found by using the following algorithm:

1. Replace $\overline{\alpha}$ with $1 - \alpha$.
2. Replace $\alpha \wedge \beta$ with $\alpha \cdot \beta$.
3. Replace $\alpha \vee \beta$ with $\alpha * \beta = 1 - (1 - \alpha)(1 - \beta)$.

This is similar to the process of arithmetization in Section 2.2.1, but the details of the two processes are different. The value of b will agree with B when $x_i \in \{0, 1\}$ for all x_i's [Shen, 878].

In order to complete the arithmetization of ϕ, we write the expression

$$\phi' = Q_1 x_1 R x_1 Q_2 x_2 R x_1 R x_2 \ldots Q_m x_m R x_1 \ldots R x_m \; B(x_1, \ldots, x_m),$$

and then rewrite ϕ' as

$$\phi' = S_1 y_1 S_2 y_2 \ldots S_k y_k \; B(x_1, \ldots, x_m),$$

where $S_i \in \{\forall, \exists, R\}$ and $y_i \in \{x_1, \ldots, x_m\}$ [Sipser, 365]. Define f_k to be b, and then for $i < k$, define f_i in terms of f_{i+1} by the following:

$$\begin{aligned}
S_i = \forall: &\quad f_i(\cdots) = f_{i+1}(\cdots, 0) \cdot f_{i+1}(\cdots, 1) \\
S_i = \exists: &\quad f_i(\cdots) = f_{i+1}(\cdots, 0) * f_{i+1}(\cdots, 1) \\
S_i = R: &\quad f_i(\cdots, a) = (1 - a) f_{i+1}(\cdots, 0) + a f_{i+1}(\cdots, 1).
\end{aligned}$$

Note that the arguments have been arranged so that the last argument is y_{i+1} and "$\cdots$" represents the values a_1 through a_j, for the appropriate value of j [Sipser, 365]. The purpose of R is to reduce the degree of the variable to one.

The protocol for deciding the truth value of ϕ is described in Figure 2.10. Note that all operations are done over the field $\mathcal{F}$, and the size of $\mathcal{F}$ must be at least d^4, where d is the length of ϕ. Refer to [Sipser, 366] or [Shen, 880] for a proof that this protocol is an interactive proof.

To illustrate the protocol we will work through a small example. Let ϕ be defined as follows:

$$\phi = \forall x_1 \exists x_2 (x_1 \vee \overline{x_2});$$

therefore,

$$\phi' = \forall x_1 R x_1 \exists x_2 R x_1 R x_2 (x_1 \vee \overline{x_2}).$$

Since $B(x_1, x_2) = x_1 \vee \overline{x_2}$,

$$\begin{aligned}
b &= x_1 * (1 - x_2) \\
&= 1 - (1 - x_1)(1 - (1 - x_2)) \\
&= 1 - (1 - x_1) x_2 \\
&= 1 - x_2 + x_1 x_2.
\end{aligned}$$

Input: A QBF ϕ and a field $\mathcal{F}$.

Phase 0: Peggy sends $f_0()$.

1. Peggy sends $f_0()$ to Vic.
2. Vic checks that $f_0() = 1$, and rejects if not.

$\vdots$

Phase i: Peggy persuades Vic that $f_{i-1}(r_1 \cdots)$ is correct, given that $f_i(r_1 \cdots)$ is correct.

1. Peggy sends the coefficents of

$$s(z) = f_i(r_1 \cdots, z),$$

where $r_1 \cdots$ is the setting of the previously selected random variables.

2. Vic checks that

$$f_{i-1}(r_1 \cdots) = \begin{cases} s(0) \cdot s(1) & \text{if } S_{i-1} = \forall \\ s(0) * s(1) & \text{if } S_{i-1} = \exists \end{cases}$$

or

$$f_{i-1}(r_1 \cdots, r) = (1-r)s(0) + rs(1) \text{ if } S_{i-1} = R.$$

If the check fails, Vic rejects; otherwise, Vic chooses $r \in \mathcal{F}$ and sends r to Peggy. If $S_{i-1} = R$, then this r replaces the previous r.

$\vdots$

Phase $k+1$: Vic checks that $f_k(r_1, \ldots, r_m)$ is correct.

Vic checks that $b(r_1, \ldots, r_m) = f_k(r_1, \ldots, r_m)$. Vic accepts if and only if they are equal.

Figure 2.10: Simplified Interactive Proof for Determining the Truth Value of a QBF [Sipser, 365-366]

Before Peggy can send f_0, she must have calculated $f_1, f_2, \ldots,$ and f_k, since each f_i depends on f_{i+1}. This is the difficult part of the protocol, and this computation is the reason Vic could not perform the protocol on his own. Before we execute the protocol, we will calculate the f_i's.

$$\begin{aligned}
f_5(x_1,x_2) &= b(x_1,x_2) \\
&= 1 - x_2 + x_1x_2 \\
f_4(x_1,x_2) &= Rx_2f_5(x_1,x_2) \\
&= (1-x_2)f_5(x_1,0) + x_2f_5(x_1,1) \\
&= (1-x_2)\cdot 1 + x_2x_1 \\
&= 1 - x_2 + x_1x_2 \\
f_3(x_1,x_2) &= Rx_1f_4(x_1,x_2) \\
&= (1-x_1)f_4(0,x_2) + x_1f_4(1,x_2) \\
&= (1-x_1)(1-x_2) + x_1 \quad 1 \\
&= 1 - x_2 + x_1x_2 \\
f_2(x_1) &= \exists x_2 f_3(x_1,x_2) \\
&= f_3(x_1,0) * f_3(x_1,1) \\
&= 1 - (1 - f_3(x_1,0))(1 - f_3(x_1,1)) \\
&= 1 - 0\cdot(1-x_1) \\
&= 1 \\
f_1(x_1) &= Rx_1f_2(x_1) \\
&= (1-x_1)f_2(0) + x_1f_2(1) \\
&= (1-x_1)\cdot 1 + x_1\cdot 1 \\
&= 1 \\
f_0() &= \forall x_1 f_1(x_1) \\
&= f_1(0)\cdot f_1(1) \\
&= 1
\end{aligned}$$

Now, we can begin the protocol. Choose $\mathcal{F}$ to be $\mathcal{Z}_{17}$. This means that all operations are performed mod17, and that every $r_i \in \mathcal{Z}_{17}$.

Phase 0:

1. Peggy sends $f_0() = 1$.
2. Vic does not reject.

Phase 1:

1. Peggy sends $s(z) = f_1(z) = 1$.
2. Since $S_1 = \forall$, Vic checks that $s(0)\cdot s(1) = f_0()$:

$$s(0)\cdot s(1) = 1\cdot 1 = 1 \equiv f_0 \bmod 17.$$

Vic chooses $r_1 = 6$.

Phase 2:

1. Peggy sends $s(z) = f_2(z) = 1$.
2. Since $S_2 = R$, Vic checks that $(1 - r_1)s(0) + r_1 s(1) = f_1(r_1)$:

$$(1-6)\cdot 1 + 6 \cdot 1 = -5 + 6 = 1 \equiv f_1(6) \bmod 17.$$

Vic chooses $r_1 = 3$, which replaces the old r_1.

Phase 3:

1. Peggy sends $s(z) = f_3(r_1, z) = 1 - z + 3z = 1 + 2z$.
2. Since $S_3 = \exists$ Vic checks that $s(0) * s(1) = f_2(r_1)$:

$$\begin{aligned} &1 - (1 - (1 + 2 \cdot 0))(1 - (1 + 2 \cdot 1)) \\ &= 1 - 0 \cdot -2 = 1 \equiv f_2(3) \bmod 17. \end{aligned}$$

Vic chooses $r_2 = 11$.

Phase 4:

1. Peggy sends $s(z) = f_4(r_1, z) = 1 - z + 3z = 1 + 2z$.
2. Since $S_4 = R$ Vic checks that $(1 - r_2)s(0) + r_2 s(1) = f_3(r_1, r_2)$:

$$\begin{aligned} &(1-11)(1+2\cdot 0) + 11 \cdot (1 + 2 \cdot 1) \\ &= -10 + 11 \cdot 3 = 23 \equiv 6 \bmod 17, \text{ and} \\ &f_3(3, 11) = 1 + 2 \cdot 11 = 23 \equiv 6 \bmod 17. \end{aligned}$$

Vic chooses $r_2 = 13$.

Phase 5:

1. Peggy sends $s(z) = f_5(r_1, z) = 1 - z + 3z = 1 + 2z$.
2. Since $S_5 = R$ Vic checks that $(1 - r_2)s(0) + r_2 s(1) = f_4(r_1, r_2)$:

$$\begin{aligned} &(1-13)(1+2\cdot 0) + 13 \cdot (1 + 2 \cdot 1) \\ &= -12 + 13 \cdot 3 = 27 \equiv 10 \bmod 17, \text{ and} \\ &f_4(3, 13) = 1 + 2 \cdot 13 = 27 \equiv 10 \bmod 17. \end{aligned}$$

Vic chooses $r_2 = 7$.

Phase 6: Vic checks that $b(3, 7) \equiv f_5(3, 7) \bmod 17$:

$$\begin{aligned} b(3,7) &= 1 - 7 + 3 \cdot 7 = -6 + 21 \equiv 15 \bmod 17, \text{ and} \\ f_5(3,7) &= 1 + 2 \cdot 7 \equiv 15 \bmod 17; \end{aligned}$$

therefore, Vic accepts Peggy's proof that ϕ is true.

Input: A QBF Q and a $p \times k$ matrix R of elements in $\mathcal{F}$.

1. Peggy prepares the superposition

$$2^{-npk/2} \sum_R |R\rangle|C(R)\rangle$$

in $\mathbf{R}$ and $\mathbf{F}$. She copies $\mathbf{R}$ to $\mathbf{S}$, and sends $\mathbf{R}$ and $\mathbf{F}$ to Vic.

2. Vic rejects if $(\mathbf{R}_i, \mathbf{F}_i)$ contains an invalid proof that Q evaluates to true, for any $i \in \{1, 2, \ldots, p\}$; otherwise, Vic chooses $u \in \{1, 2, \ldots, k\}^p$ uniformly at random, and sends u and $\overline{\mathbf{F}}^{(u)}$ to Peggy.

3. Peggy applies $T_{i,j}^{-1}$ to $\mathbf{S}$ together with $\mathbf{F}_{i,j}$, for each appropriate pair i, j (this returns the registers of $\overline{\mathbf{F}}^{(u)}$ to their initial zero value). Peggy sends $\mathbf{S}$ to Vic.

4. Vic subtracts $\mathbf{R}_{i,j}$ from $\mathbf{S}_{i,j}$ for each i, j. He then applies transform $H^{\otimes K}$ to each register of $\overline{\mathbf{R}}^{(u)}$. If it now only contains zeros, then accept; otherwise, reject.

Figure 2.11: Quantum Interactive Proof for Determining the Truth Value of a QBF [Watrous, 7]

2.3.3 Quantum Interactive Proof for QBF

The quantum interactive proof for proving the truth value of a QBF is shown in Figure 2.11. Note that m, k, d, and $\mathcal{F}$ are as defined in Section 2.3.2 and in the protocol in Figure 2.10. The capital, boldface letters represent **registers**, which are collections of qubits upon which transformations are performed. This protocol uses registers $\mathbf{R}_{i,j}, \mathbf{S}_{i,j}$, and $\mathbf{F}_{i,j}$, $1 \leq i \leq p$ and $1 \leq j \leq k$, where p is some polynomial in m and $k = \binom{n+1}{2} + n$. Each $\mathbf{R}_{i,j}$ and $\mathbf{S}_{i,j}$ is a collection of n qubits, where 2^n is the size of the field $\mathcal{F}$ that the polynomials are taken to be over [Watrous, 6]. The classical states of these registers are elements of $\mathcal{F}$.

Each $\mathbf{F}_{i,j}$ is made up of $d+1$ collections of n qubits, where d is the length of the QBF, and the classical states of these registers are polynomials of degree at most d and whose coefficents are in $\mathcal{F}$ [Watrous, 6]. Given $u \in \{1, 2, \ldots, k\}^p$, $\mathbf{R}^{(u)}$ refers to the collection of registers $\mathbf{R}_{i,1}, \mathbf{R}_{i,2}, \ldots, \mathbf{R}_{i,u_i-1}$ and $\mathbf{F}^{(u)}$ refers to $\mathbf{F}_{i,1}, \mathbf{F}_{i,2}, \ldots, \mathbf{F}_{i,u_i}$ for $i \in \{1, 2, \ldots, p\}$ [Watrous, 6]. Refer to Figure 2.12 for an example of this.

$C(R)$ is a matrix of polynomials such that for all $i \in \{1, 2, \ldots, p\}$, $C(R)_{i,1}$, $C(R)_{i,2}, \ldots, C(R)_{i,N}$ is the sequence of polynomials an honest prover returns, in the protocol described in Section 2.3.2, given the random numbers $R_{i,1}, R_{i,2}, \ldots$, $R_{i,k}$ [Watrous, 7]. In other words, in step 1, Peggy prepares each row of the matrix $C(R)$ so that it contains the polynomials $f_1, f_2, \ldots, f_k$ from the protocol

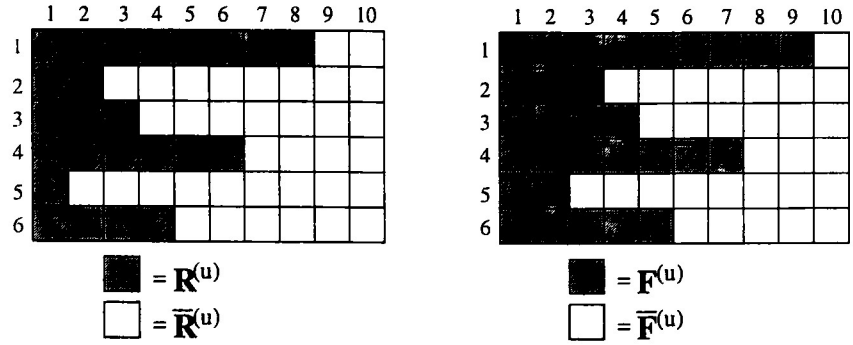

Figure 2.12: Example division of $\mathbf{R}$ and $\mathbf{F}$ for $N = 10$, $m = 6$, and $u = (9, 3, 4, 7, 2, 5)$ [Watrous, 6].

in Figure 2.10. Then, in step 2, Vic checks that each row of polynomials forms a valid proof of the truth value of the QBF. If any of the rows do not form valid proofs, he rejects Peggy's proof. Vic needs to use the gate for reversible computation to perform the check in step 2.

$T_{i,j}$ is the unitary transformation defined by:

$$T_{i,j} : |\mathbf{R}\rangle|0\rangle \rightarrow |\mathbf{R}\rangle|C(R)_{i,j}\rangle \text{ [Watrous, 7].}$$

$H^{\otimes K}$ is the Walsh-Hadamard transform, where

$$H : |0\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \text{ and } H : |1\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \text{ [Watrous, 6].}$$

Refer to [Watrous] for a proof of completeness and soundness of this protocol. The error probability of the protocol is determined by p and n, and they are assumed to have values so that the error is exponentially small [Watrous, 6]. Watrous uses this protocol to prove the following theorem:

Theorem 2.12 *Every language in PSPACE has a 2-round quantum interactive proof system with exponentially small error [Watrous, 2].*

Quantum interactive proofs appear to be more powerful than classical interactive proofs. All known classical interactive protocols for PSPACE problems involve a non-constant number of rounds, and they cannot be parallelized, but by Theorem 2.12 there exists a constant round quantum interactive protocol for every PSPACE problem. Refer to [Babai] or [Goldwasser92] for the proof that if the polynomial hierarchy (PH) is non-collapsing, then there does not exist constant round classical interactive protocols for PSPACE problems.

Chapter 3

Perfect Zero-Knowledge Proofs

An interactive proof is perfect-zero knowledge if the only thing that is learned by the verifier of the proof is the outcome of the proof. In terms of the discussion of interactive proofs, this means that Vic learns nothing from Peggy that he could not have learned by himself, except the fact that x is a yes or no-instance of the decision problem. This means that Vic is completely convinced of what Peggy is proving, but he could not prove the decision problem to anyone else so that they would be convinced.

To prove that an interactive proof is perfect zero-knowledge, Vic needs to be able to forge a transcript of the proof in a non-interactive way, since the only way to show that Vic has learned nothing from Peggy is for him to reproduce the proof by himself. A transcript of an interactive proof is a listing of all inputs to the proof and all messages sent or received by Vic during the proof. If Vic can generate a forged transcript that resembles the real transcript, without interacting with Peggy, then the proof is zero-knowledge for Vic. A transcript resembles another transcript if the probability distributions of the two transcripts are identical.

In this scheme to prove zero-knowledge, it is assumed that Vic will not deviate from the protocol. This is why the proof is only considered zero-knowledge for Vic. What happens if he does cheat? Is the proof still zero-knowledge? To prove that the proof is unconditionally zero-knowledge, Vic must be able to create an algorithm that, given any cheating scheme, will create a forged transcript that resembles the real transcript for that cheating scheme.

3.1 Quadratic Residues

The proof presented in Figure 2.1 is not perfect zero-knowledge since Vic has no way of creating a forging algorithm. He would need to generate the triples (v, i, j), and given any two of these numbers and the fact that he is restricted

Input: An integer n, with unknown factorization pq, where p and q are prime, and x, a quadratic residue of n.

1. Repeat the following steps $\log_2 n$ times:

   (a) Peggy chooses $v \in \mathcal{Z}_n^*$ at random, and she then computes

   $$y = v^2 \bmod n.$$

   Peggy sends y to Vic.

   (b) Vic chooses $i \in \{0, 1\}$ at random, and he sends i to Peggy.

   (c) Peggy computes

   $$z = u^i v \bmod n,$$

   where $u \in \mathcal{Z}_n^*$ and is a solution to $u^2 \equiv x \bmod n$. Peggy sends z to Vic.

   (d) Vic checks that

   $$z^2 \equiv x^i y \bmod n.$$

2. Vic accepts Peggy's proof, that x is a quadratic residue of n, if step (d) is verified in each of the $\log_2 n$ rounds.

Figure 3.1: Perfect Zero-Knowledge Proof for Quadratic Residues [Stinson, 396]

to polynomial time algorithms, he has no way of generating the third number so that the triples will have the same probabilities as they did in the original proof. An example of a perfect zero-knowledge proof is the protocol for quadratic residues given in Figure 3.1.

3.1.1 Perfect Zero-Knowledge Proof for Quadratic Residues

The following theorem proves that the protocol for quadratic residues given in Figure 3.1 is an interactive proof by showing completeness and soundness.

Theorem 3.1 *The protocol for quadratic residues is complete and sound.*

Proof: To show that the proof is complete, show that if x is a quadr atic residue then Vic will always accept Peggy's proof. Assume that this is not true: x is a quadratic residue, and Vic rejects Peggy's proof. This will only occur if $z^2 \not\equiv x^i y \bmod n$ in at least one of the $\log_2 n$ rounds. Since x is a quadratic residue, there exists a $u \in \mathcal{Z}_n^*$ which is a solution to $u^2 \equiv x \bmod n$; therefore,

$$\begin{aligned} z &= u^i v \bmod n, u^2 \equiv x \bmod n \text{ and } y = v^2 \bmod n \\ z^2 &\equiv (u^i v)^2 \bmod n \\ &\equiv u^{2i} v^2 \bmod n \end{aligned}$$

Input: An integer n with unknown factorization pq, where p and q are prime, and x, a quadratic residue of n.

1. $\mathcal{T} = (n, x)$
2. For $j = 1$ to $\log_2 n$
   (a) Choose $i_j \in 0, 1$ at random.
   (b) Choose $z_j \in \mathcal{Z}_n^*$ at random.
   (c) Compute $y_j = z_j^2 (x^i)^{-1} \bmod n$.
   (d) Concatenate the triple (y_j, i_j, z_j) onto the end of $\mathcal{T}$.

Figure 3.2: Forging Algorithm for Transcripts for Quadratic Residues

$$\equiv \quad x^i y \bmod n.$$

This is a contradiction, so our assumption is false; the protocol for quadratic residues is complete.

To prove soundness, show that if x is a quadratic non-residue of n, then there is only a small probability that Vic will accept Peggy's proof. If x is a quadratic non-residue, then Peggy will not be able to calculate u in step c, and the proof will only work if for all $\log_2 n$ rounds if Vic chooses $i = 0$ in every round, so that the exponent on u is zero. The probability of this occurring, since i is chosen at random from {0, 1}, is $\frac{1}{2^{\log_2 n}}$ or $1/n$. This means that for large n the proof is sound.

To show that the protocol for quadratic residues is a zero-knowledge proof, there needs to be a valid forging algorithm. The forging algorithm is shown in Figure 3.2. The output of the forging algorithm is a transcript $\mathcal{T}$, consisting of the input values n and x and triples (y, i, z) for each round. Theorem 3.2 proves that the protocol for quadratic residues is a perfect zero-knowledge proof for Vic by showing the validity of this forging algorithm; it shows that the probability of a triple being generated by the forging algorithm is the same as the probability of the triple being generated by the original proof.

Theorem 3.2 *The protocol for quadratic residues is perfect zero-knowledge for Vic.*

Proof: To show that the protocol for quadratic residues is perfect zero-knowledge, it needs to be verified that given the input to the proof, the forging algorithm and the original proof create transcripts with identical probability distributions. In the original proof, the triple is uniquely determined by the selection of i and v. i is chosen at random from {0, 1}; therefore, there are two choices for i, each

with equal probability. v is chosen randomly from $\mathcal{Z}_n^*$ which has $(p-1)(q-1)$ elements; therefore, there are $(p-1)(q-1)$ choices for v, again each with equal probability. The probability of any given triple being calculated is

$$[2(p-1)(q-1)]^{-1}.$$

In the forging algorithm, i is chosen at random from $\{0, 1\}$, and z is chosen from $\mathcal{Z}_n^*$ randomly. These two numbers exactly determine y; therefore, the probability of any given triple being calculated by the forging algorithm is the same as the probability of it being calculated by the original proof. This means that the protocol for quadratic residues in Figure 3.1 is perfect zero-knowledge for Vic.

To show that the proof is unconditionally zero-knowledge, it needs to be shown that a forging algorithm can be created for any cheating version of Vic. This can be done, but the details of that proof will not be shown here. Refer to [Stinson, 393-395] for a description of this method.

3.1.2 Implementation of Protocol for Quadratic Residues

The algorithm in Figure 3.1 is implemented by the program `quadres`. `quadres` is composed of two subprograms which implement Peggy and Vic. The programs are run simultaneously, in separate windows, in order to simulate the interaction between Peggy and Vic.

`quadres` is more restricted than the algorithm it implements since in principle Peggy is not restricted to polynomial time, but in `quadres` she must work in polynomial time. To help compensate for this shortcoming, the program allows the user several options. One choice the user has is deciding which method Peggy will use to choose what she is going to prove. The options are:

Method 1 Peggy randomly chooses n and u, and then calculates

$$x = u^2 \bmod n.$$

Method 2 Peggy randomly chooses n and x, and then finds u if it exists.

Method 3 Peggy reads in a random entry from a file of fifty entries, where an entry consists of the three numbers n, x, and u.

Method 1 is easily the fastest method, but it will always result in x's that are quadratic residues of n. The input chosen by Methods 2 and 3 do not require that x be a quadratic residue of n, but, for Method 2 and possibly for Method 3, depending on how the file is generated, the purpose of the proof is being undermined. The proof that the algorithm in Figure 3.1 is perfect zero-knowledge relies on the assumption that finding the square root of a number modulo n is hard. That calculation is what Peggy must do in Method 2. Because

Peggy must do a non-polynomial time calculation, the number of digits in n is severely restricted.

Vic's other choices will only effect Peggy's choice of n. Vic can choose whether or not n is required to be the product of two primes. The idea behind adding this choice was that the calculation might be faster if it was only necessary to find one large random number rather than finding two large random primes, but this has not turned out to be true. Since there is no major difference in time, it is better to choose n to be the product of two primes, since the protocol is not secure if n is made up of small factors. Vic can also choose if n should always have the maximum number of digits, or if n should have a random number of digits. The maximum number of digits is set to two hundred.

The output of `quadres` is both a transcript written to the screen and a transcript written to a file. The transcript written to the screen consists of a detailed description of the inputs to the program, the messages between Peggy and Vic, and the decision made by the program. The transcript is in two parts corresponding to what Peggy knows and to what Vic knows. This helps the user visualize what information is private to either Peggy or Vic. The transcript that is written to a file is appended to the file `quadres.trans`, and it consists of only n and x, y, i, and z from each round, and either the word SUCCESS or FAILURE, depending on the outcome of the program. The transcript is saved this way so that it is comparable to the transcript of the forging algorithm.

The forging algorithm in Figure 3.2 is implemented by the program `qrforge`. The inputs to the program are simply n and x, and the output from the program is a file `qrforge.trans`, which has the same format as `quadres.trans`. Since the inverse of x is needed in the forging algorithm, n and x must be relatively prime. If n is the product of two large primes, it is very unlikely for n and x to have common factors, since x would have to be one of the two prime factors of n. If n is small, or if n is not the product of two primes, it is more likely that n and x will have a common factor. This is another reason for choosing n to be the product of two primes, and for using Method 1.

Several test runs of `quadres` and `qrforge` can be found in Appendix B.2. n should have at least a few hundred digits so that the protocol will be secure.

3.2 Subgroup Membership

The protocol for subgroup non-membership is similar to the protocol for quadratic non-residues; it is not perfect zero-knowledge since no forging algorithm is known. Figure 3.3 presents a protocol for subgroup membership.

3.2.1 Perfect Zero-Knowledge Proof for Subgroup Membership

The proof shown in Figure 3.3 is perfect zero-knowledge for Vic as shown by Theorem 3.3, which refers to the forging algorithm presented in Figure 3.4.

Input: A positive integer n and two distinct elements $a, b \in \mathcal{Z}_n^*$, where the order of a is denoted by m and is publicly known.

1. Repeat the following steps $\log_2 n$ times:

   (a) Peggy chooses $j \in \{0, 1, \ldots, m-1\}$ at random and computes

   $$g = a^j \bmod n,$$

   and sends g to Vic.

   (b) Vic chooses $i \in \{0, 1\}$ at random. He sends i to Peggy.

   (c) Peggy computes

   $$h = j + ik \bmod m,$$

   where $k \in \{0, 1, \ldots, m-1\}$, such that $b \equiv a^k \bmod n$. She sends h to Vic.

   (d) Vic checks that $a^h \equiv b^i g \bmod n$.

2. Vic accepts Peggy's proof, that b is a member of the subgroup of $\mathcal{Z}_n^*$ generated by a, if step d is verified in each of the $\log_2 n$ rounds.

Figure 3.3: Perfect Zero-Knowledge Proof for Subgroup Membership [Stinson, 398]

Theorem 3.3 *The protocol for subgroup membership is perfect zero-knowledge for Vic.*

Proof: To show that the protocol for subgroup membership is perfect zero-knowledge, it needs to be verified that the proof is an interactive proof, in other words, that it is complete and sound, and that there exists a valid forging algorithm.

To prove that the protocol for subgroup membership is complete, show that if b is a member of the subgroup of $\mathcal{Z}_n^*$ generated by a, then Vic always accepts Peggy's proof. Vic will accept Peggy's proof if for all $\log_2 n$ rounds, $a^h \equiv b^i g \bmod n$. Since b is a member of the subgroup generated by a, there exists a $k \in \{0, 1, \ldots, m-1\}$ such that $b \equiv a^k \bmod n$; therefore,

$$\begin{aligned} h &= j + ik \bmod m, g = a^j \bmod n \text{ and } b \equiv a^k \bmod n \\ a^h &\equiv a^{j+ik} \bmod n \\ &\equiv (a^k)^i a^j \bmod n \\ &\equiv b^i g \bmod n. \end{aligned}$$

This shows that the proof is complete.

Input: A positive integer n and two distinct elements $a, b \in Z_n^*$ where the order of a is denoted by m and is publicly known.

1. $\mathcal{T} = (n, a, b, m)$
2. For $k = 1$ to $\log_2 n$
   (a) Choose $h_k \in \{0, 1, \ldots, m-1\}$ at random.
   (b) Choose $i_k \in \{0, 1\}$ at random.
   (c) Compute $g_k = a_k^h (b_k^i)^{-1} \bmod n$.
   (d) Concatenate the triple (g_k, i_k, h_k) onto the end of $\mathcal{T}$.

Figure 3.4: Forging Algorithm for Transcripts for Subgroup Membership

The protocol for subgroup membership is sound if b is not a member of the subgroup generated by a, then there is only a small probability that Vic will accept Peggy's proof. Vic accepts Peggy's proof if for all $\log_2 n$ rounds, $a^h \equiv b^i g \bmod n$. Since $g = a^j \bmod n$, this is only true if $b \equiv a^k \bmod n$, for some $k \in \{0, 1, \ldots, m-1\}$, or if $i = 0$. The first case is a contradiction, since it was assumed that there does not exist such a k; therefore, the only way Vic will accept Peggy's proof is if $i = 0$ in all rounds. The probability of this occurring, since i is chosen at random from {0, 1}, is $\frac{1}{2^{\log_2 n}}$ or $1/n$. This means that for large n the proof is sound.

To complete the proof, it needs to be shown that the forging algorithm given in Figure 3.4 is valid. The output of both the original proof and the forging algorithm is a transcript $\mathcal{T}$, which consists of the inputs, n, a, b, and m, and a triple, (g, i, h), for each of the $\log_2 n$ rounds. In the original proof, each triple is uniquely determined by the choices of j and i. There are m choices for j, each with equal probability, since j is chosen at random from $\{0, 1, \ldots, m-1\}$. There are two choices for i, also each with equal probability, since i is chosen at random from 0, 1. This gives a probability of $1/(2m)$ for the calculation of any given triple. The forging algorithm exactly determines the triple by selecting h and i. The selections of these two variables are analogous to the selection of j and i in the original proof; therefore, the probability of calculating a given triple is the same for both the forging algorithm and the original proof. This completes the proof. The protocol for subgroup membership is perfect zero-knowledge for Vic.

3.2.2 Implementation of Protocol for Subgroup Membership

The program `subgrp` implements the perfect zero-knowledge protocol for subgroup membership, given in Figure 3.3. `subgrp` has the same format as `quadres`; it is split into two subprograms, one representing Peggy and one representing Vic.

As in `quadres`, there are several methods Peggy can use to choose what she is going to prove:

Method 1 Vic inputs the values of n, a, and m such that

$$a^m \equiv 1 \bmod n.$$

Peggy then chooses k at random and calculates b such that

$$b = a^k \bmod n.$$

Method 2 Peggy chooses n, a, and b at random, such that a and b are relatively prime to n. She then calculates k and finds m, if it exists.

Method 3 Peggy reads a random entry from a fifty-entry file, where an entry consists of the five numbers n, a, m, k, and b.

These methods are similar to the methods for `quadres`, so therefore, they have similar properties. Method 1 is always going to give b's that are members of the subgroup of $\mathcal{Z}_n$ generated by a, whereas Methods 2 and 3 will give b's that sometimes are and sometimes are not members of the subgroup. Method 1 can perform with large numbers of digits, but Method 2 is very slow even for small numbers of digits. Vic also has the same choices about n as he did for `quadres`. Note that there is no efficient way to generate examples of subgroup members for large n's, since when using Method 1, Vic is required to provide n, a, and m. Method 1 was implemented this way because there is no efficient algorithm for computing the order of a mod n.

The output of `subgrp` is both the transcript printed to the screen and one printed to a file. The transcript printed to the file is appended to `subgrp.trans` and contains n, a, m, and b, g, i, and h from each round, and either `SUCCESS` or `FAILURE` depending on the outcome of the proof. The forging algorithm in Figure 3.4 is implemented by the program `sgforge`. The inputs to this program are n, a, m, and b, and the transcript from this program is saved to `sgforge.trans`. m is required to be the order of a modulo n, and b and n are required to be relatively prime. Several test runs of `subgrp` and `sgforge` can be found in Appendix B.3. Note that n should have at least a few hundred digits so that the protocol will be secure.

Chapter 4

History and Applications

Several references contain overviews of the history of interactive and zero-knowledge protocols. Johnson gives an extremely detailed overview of the early history in his NP-completeness column [Johnson]. This paper includes several ideas for applications of these protocols. Some cryptography textbooks, including [Stinson, Ch. 13] and [Menezes, 405-417,421-424], have a thorough description of the various definitions of zero-knowledge and interactive protocols. For an entertaining presentation, refer to [Quisquater], which uses a discussion of the secrets of Ali Baba's cave to give a simple illustration of zero-knowledge protocols.

The notion of an interactive proof system was independently developed by Goldwasser et al. [Goldwasser85] and by Babai [Babai]. The two definitions are slightly different, but they were proved to be equivalent by Goldwasser and Sipser [Goldwasser92]. The version discussed in this thesis is the version that was introduced in [Goldwasser85].

Babai calls his version of an interactive proof systems an "Arthur-Merlin game". In this definition, Peggy goes by the name Merlin, and Vic goes by the name Arthur. Merlin's job in the protocol is exactly the same as Peggy's, but Arthur is more restricted than Vic. His messages may only consist of sequences of random bits obtained from his private source, where the length of the sequences is determined only by the input to the protocol. Because of this restriction Arthur has nothing to do until Merlin sends his final message, then Arthur has polynomial time to accept or reject Merlin's proof based on the input and the transcript [Johnson, 430]. [Goldwasser89] is a final version of [Goldwasser85] which includes a description of Arthur-Merlin games.

Shamir's proof that IP = PSPACE [Shamir] is an extension of a result found by Lund et al. [Lund]. They proved that IP contains the polynomial hierarchy, PH. Prior to this, Golgreich et al. had shown that IP contained some languages which were not believed to be in NP [Goldreich]. In particular, they present an IP for graph non-isomorphism. It was also shown in [Goldreich] that under the assumption of the existence of secure encryption functions, all languages in NP have zero-knowledge proofs.

Several generalizations of zero-knowledge proofs have been created, including computational zero-knowledge proofs and zero-knowledge arguments. For perfect zero-knowledge the forging algorithm must create transcripts which have the same probability distributions as transcripts created by the original proof. This condition can be relaxed to create computational zero-knowledge proofs. All that is required for computational zero-knowledge is for the two transcripts to be indistinguishable in polynomial time, rather than having the same probabilistic distributions. Bit commitment schemes are often used to create these proofs.

The following is a description of how a bit commitment scheme works. Peggy chooses a message and places it in a safe. Only she holds the key to the safe. Peggy is committed to the message since she can not change what is in the safe. Vic can not know what is in the safe unless Peggy opens the safe for him, or she gives him the key to the safe. The message that Peggy places in the safe is a bit. The bit commitment scheme is a function $f : \{0,1\} \times \mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are finite sets. Peggy chooses $x \in \mathcal{X}$ and encrypts her message, the bit she wishes to conceal, into a blob using f. The blob is an element of $\mathcal{Y}$.

A valid bit commitment scheme must be concealing and binding. The scheme is concealing if Vic can not determine from the blob what the original bit was, and the scheme is binding if Peggy can not unlock the blob as both a zero and a one. In a computational zero-knowledge proof, the concealing condition can depend upon a computational assumption and the binding condition must be unconditional. Zero-knowledge arguments use bit-commitment schemes which have a computational assumption for the binging condition and have an unconditional concealing condition. Computational zero-knowledge proofs can be transformed into zero-knowledge arguments by changing the bit commitment scheme that is used in the protocol. The concepts of computational zero-knowledge proofs, zero-knowledge arguments, and bit-commitment schemes are discussed in [Stinson, 398-407]. For a more detailed discussion refer to [Brassard88]. The term zero-knowledge argument was first introduced in [Brassard90], which also contains a user-friendly discussion of the various definitions of zero-knowledge.

One clear application of interactive and zero-knowledge protocols is that any problem in NP can be proven by a zero-knowledge protocol, and this implies that any mathematical proof can be converted to a zero-knowledge protocol [Schneier, 108]. This can be used to taunt your colleagues, since you can prove to them that you know of the existence of a proof to a certain problem without revealing the details of that proof. A more useful application is the notion of a proof of identity. A proof of identity allows Vic to verify Peggy's identity without Peggy having to reveal any information, such as a password. A simple example of this type of protocol is given in [Salomaa], and more information can be found in [Simmons] and [Feige]. Proofs of identity can be used in the development of smart cards. Refer to [Simmons, 599-603] or [Seberry, 296-298] for a detailed description of this application. Zero-knowledge concepts can also be used for playing cards over a network. There are several papers which discuss poker protocols, but, in particular, Crépeau has focused on a zero-knowledge poker protocol in [Crépeau].

Appendix A

Implementation

All programs were written in C++. `qbf` uses two classes, `Polynomial` and `Formula`. Both of these classes implement all the operations needed in `qbf`, as well as doing their own input and output. The programs `quadres`, `qrforge`, `subgrp`, and `sgforge` share an include file `zero_knowledge.h`, which includes the necessary files, declares the common functions, and has the necessary macro definitions.

The Multiple Precision Integer and Rational Arithmetic C Library (MIRACL) was used to implement multi-precision arithmetic. Norman Moulton's modifications to the library, as described in [Moulton], were used so that the library worked on the R.I.T. Sun workstations. The MIRACL package includes a C++ class `Big` which implements multi-precision integers. `Big`'s can be used just as integers since all of the operators have been overloaded. The same principle was used in developing the classes `Polynomial` and `Formula`. The MIRACL library is available on the internet at [MIRACL]. It is a shareware package that is free for academic use.

To use the MIRACL package, the library `miracl.a` must first be compiled. Before this is done, the file `mirdef.h` needs to be modified for the specific hardware that you are using. This process is described in the manual provided with MIRACL. The package includes both rational and integer arithmetic, but if the rational arithmetic package is not needed, then the library can be compiled without it. Note that MIRCAL uses probabilistic primes rather than pure-math primes. The following is the copy of `mirdef.h` that I used to compile MIRACL on the R.I.T. Sun workstations:

```
#define MIRACL 32
#define mr_utype int
#define mr_unsign32 unsigned int
#define mr_dltype long long
#define MR_IOBSIZ 5000
#define MR_NOASM
#define MR_NOFULLWIDTH
```

Once the library has been compiled, simply copy the files `miracl.a`, `miracl.h`, `mirdef.h`, `big.h`, and `big.cpp` to the directory where your code is located, and compile your code with the library `miracl.a`. See the sample program below for an example of how to use MIRACL:

```
//
// Name:        sample.cpp
// Author:      Molli Noland
// Date:        May 2, 1999
// Description: A sample program for demonstrating how to use
//              the MIRACL
//              library and the Big class.
//

#include <iostream.h>
#include "big.h"

Miracl precision( 300, 10 );  // 300 decimal digits

main(){
  Big   x, y, n, p;

  cout << "Enter a LARGE number x: ";
  cin >> x;
  cout << "Enter a LARGE number y: ";
  cin >> y;
  cout << "Enter a LARGE number n: ";
  cin >> n;

  cout << "\n\nx + y = " << x + y << endl;
  cout << "x * y = " << x * y << endl;

  if( x > y ) cout << "x mod y = " << x % y << endl;
  else cout << "y mod x = " << y % x << endl;

  cout << "x^y mod n = " << pow(x,y,n) << endl;

  p = nextprime( n );
  cout << "The first prime after n is: " << p << endl;
}
```

Running sample, the following output is obtained:

```
Enter a LARGE number x: 217863486493658648986217563462142784847693
Enter a LARGE number y: 725875372578352757821582769849479570477
Enter a LARGE number n: 56526754754835781758757575475378 4
```

```
x + y = 217936074030916484261999721739127732804740

x * y = 158141739429803395716120726091615439401213328606390164
247930877180144874010425 71

x mod y = 28287182894986363960574230313965749646

x^y mod n = 338677434042267349688787062476501

The first prime after n is: 56526754754835781758757575475384 7
```

It should be noted that there is an error in the code of `subgrp` and `sgforge`, that occurs only when n is an odd multiple of five. The error actually occurs in a library function, and the error message says, in reference to n, "Illegal Montgomery Modulus (must be odd)". The documentation for the library referred to [Montgomery], but since the value of n is odd when this error is occurring, no solution, other than not allowing n to be an odd multiple of five, was found for this problem. This problem actually does not come up at all if n is chosen to be a multiple of two primes and n has more than three digits.

Appendix B

Test Runs of Programs

The following sections include the sample input and output of the programs: `qbf`, `quadres`, `qrforge`, `subgrp`, and `sgforge`.

B.1 Quantified Boolean Formulas (qbf)

The program `qbf` implements the protocol for determining the truth value of a QBF in Figure 2.3. `qbf` uses the classes `Polynomial` and `Formula` to implement the protocol. `Polynomial` implements polynomials of a single variable, and objects of this type are used to represent the polynomials $q(z_i)$. `Formula` implements QBF's and all the operations performed on them in the protocol.

The first sample input to `qbf` is the QBF

$$Q = \forall x_1 \exists x_2 (x_1 \wedge \overline{x_2}).$$

This is the sample that was worked through in Section 2.2.2. The input file for Q is:

```
A x1 E x2 ( x1 & ! x2 )
```

It is important to note that there can be no extra whitespace at the end of the input file. When `qbf` was run on Q, the following output was obtained:

```
Peggy needs a seed before she can choose random numbers.
Number: 124

Input:
-----
        A: Ax1 Ex2 ( x1 & !x2 )
        p: 17
```

```
Peggy:
-----
        a (mod p): 0
        A1:
        A2: Ax1 Ex2 ( x1 & !x2 )
        A = A2

Vic:
---
        I need more information!!

Peggy:
-----
        A': Ex2 ( x1 & !x2 )
        q(x1): x1

Vic:
---
        A = A'(x1 = 6): Ex2 ( 6 & !x2 )
        a: 6

Peggy:
-----
        a (mod p): 6
        A1:
        A2: Ex2 ( 6 & !x2 )
        A = A2

Vic:
---
        I need more information!!

Peggy:
-----
        A': 6 & !x2
        q(x2): -6x2 + 6

Vic:
---
        A = A'(x2 = 1): 6 & 0
        a: 0

Peggy:
-----
        a (mod p): 0
        A1: 6 & 0
```

```
        A2:
        A = A1

Vic:
---

        The value of Ax1 Ex2 ( x1 & !x2 )  IS 0 mod 17
```

Since Vic accepts Peggy's proof, the QBF Q is false, which was shown in Section 2.2.2.

Now that we have seen a sample where Vic accepts, we will turn to a case where he rejects Peggy's proof. The following is the output from the QBF:

$$R = \forall x_1 \exists x_2 (x_1 \vee \overline{x_2})$$

```
Peggy needs a seed before she can choose random numbers.
Number: 1884

Input:
-----
        A: Ax1 Ex2 ( x1 | !x2 )
        p: 51739568706385042681087936474916914105687443338150168
16301112758686156410420987129658294864191918861

Peggy:
-----
        a (mod p): 0
        A1:
        A2: Ax1 Ex2 ( x1 | !x2 )
        A = A2

Vic:
---
        I need more information!!

Peggy:
-----
        A': Ex2 ( x1 | !x2 )
        q(x1): 2x1 + 1

Vic:
---

        The value of Ax1 Ex2 ( x1 | !x2 )  IS NOT 0 mod 51739568
70638504268108793647491691410568744333815016816301112758686156 41
04209871296582948641919188 61
```

Since Vic rejected Peggy's proof, the QBF R is true.

These examples were simple. To see a more complex example, we will use the test described in Section 2.2.4. The QBF is

$$B = \exists x_1 \forall x_2 \exists x_3 \forall x_4 \exists x_5 ((\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_1 \vee x_4 \vee x_5) \wedge (x_3 \vee \overline{x_4} \vee \overline{x_5})),$$

but since the input to `qbf` must be a simple QBF, B needed to be rewritten so that it was simple; therefore, the input file for B is:

```
E x1 A x2 E x1prime ( ( ( x1prime & x1 ) | ( ! x1prime & ! x1 )
) & ( E x3 A x4 E x5 ( ( ( ( ! x1prime | ! x2 ) | x3 ) & ( ( x2
| ! x3 ) | ! x4 ) ) & ( ( ( x1prime | x4 ) | x5 ) & ( ( x3 | !
x4 ) | ! x5 ) ) ) ) )
```

The following is the output from `qbf` when run on B:

```
Peggy needs a seed before she can choose random numbers.
Number: 790

Input:
-----
        A: Ex1 Ax2 Ex1prime ( ( ( x1prime & x1 ) | ( !x1prime &
!x1 ) ) & ( Ex3 Ax4 Ex5 ( ( ( ( !x1prime | !x2 ) | x3 ) & ( ( x2
| !x3 ) | !x4 ) ) & ( ( ( x1prime | x4 ) | x5 ) & ( ( x3 | !x4 )
| !x5 ) ) ) ) )
        p: 588588169804627409208736429934879882143740957133192487
34777391529385509569782004549334353872836093 21

Peggy:
-----
        a (mod p): 0
        A1:
        A2: Ex1 Ax2 Ex1prime ( ( ( x1prime & x1 ) | ( !x1prime &
!x1 ) ) & ( Ex3 Ax4 Ex5 ( ( ( ( !x1prime | !x2 ) | x3 ) & ( ( x2
| !x3 ) | !x4 ) ) & ( ( ( x1prime | x4 ) | x5 ) & ( ( x3 | !x4 )
| !x5 ) ) ) ) )
        A = A2

Vic:
---
        I need more information!!

Peggy:
-----
        A': Ax2 Ex1prime ( ( ( x1prime & x1 ) | ( !x1prime & !x1
) ) & ( Ex3 Ax4 Ex5 ( ( ( ( !x1prime | !x2 ) | x3 ) & ( ( x2 | !
x3 ) | !x4 ) ) & ( ( ( x1prime | x4 ) | x5 ) & ( ( x3 | !x4 ) |
```

```
!x5 ) ) ) ) )
        q(x1): 224x1^2 + 784x1 + 560

Vic:
---

        The value of Ex1 Ax2 Ex1prime ( ( ( x1prime & x1 ) | (
!x1prime & !x1 ) ) & ( Ex3 Ax4 Ex5 ( ( ( ( !x1prime | !x2 ) | x3
) & ( ( x2 | !x3 ) | !x4 ) ) & ( ( ( x1prime | x4 ) | x5 ) & ( (
x3 | !x4 ) | !x5 ) ) ) ) )  IS NOT 0 mod 58858816980462740920873
64299348798821437409571331924873477739152938550956978200454933 43
5387283609321
```

Since Vic rejected Peggy's proof, the QBF B is true. Recall that the consequence of this QBF being true is that the graph in Figure 2.8 can be pebbled by eighteen pebbles. As mentioned before, instances of QBF's which may lead to applications should have at least 100 variables and 400 clauses.

B.2 Quadratic Residues

The program `quadres` implements the zero-knowledge protocol for quadratic residues in Figure 3.1, and the program `qrforge` implements the corresponding forging algorithm in Figure 3.2.

B.2.1 quadres

The program `quadres` is a simple shell script which runs the programs `quadres_peggy` and `quadres_vic` in separate windows. `quadres_peggy` and `quadres_vic` represent Peggy and and Vic's separate parts of the protocol. The two programs communicate throughout the protocol by reading and writing to a set of shared files.

To demonstrate how the program `quadres` works, we will look at a simple example. Even though on the screen Vic's window would be on the left and Peggy's window would be on the right, here Vic's output is listed first, and then Peggy's output. `quadres_vic` will prompt the user for information, including a seed to start the random number generation and answers to a series of questions. Vic's output is as follows:

```
Peggy needs a seed before she can decide what she will prove.
Number: 7
Which decision method is to be used? (1, 2, or 3): 1
Should n be the product of two primes? (y/n): y
Should n always contain the maximum number of digits? (y/n): n
Peggy sent n = 1411
           x = 1339
The transcript will be appended to the file 'quadres.trans'
```

```
------------------------------------------------------------
                Begin Round 1
------------------------------------------------------------
Peggy sent y = 400
I set i = 1
Peggy sent z = 248
My check SUCCEEDED:
        z^2 = 61504
                IS congruent to
        x^i*y = 535600
                modulo 1411
------------------------------------------------------------
                End Round 1
------------------------------------------------------------
------------------------------------------------------------
                Begin Round 2
------------------------------------------------------------
Peggy sent y = 562
I set i = 1
Peggy sent z = 1096
My check SUCCEEDED:
        z^2 = 1201216
                IS congruent to
        x^i*y = 752518
                modulo 1411
------------------------------------------------------------
                End Round 2
------------------------------------------------------------

 .
 .
 .

------------------------------------------------------------
                Begin Round 11
------------------------------------------------------------
Peggy sent y = 64
I set i = 1
Peggy sent z = 1228
My check SUCCEEDED:
        z^2 = 1507984
                IS congruent to
        x^i*y = 85696
                modulo 1411
------------------------------------------------------------
                End Round 11
------------------------------------------------------------
1339 IS a quadratic residue modulo 1411
```

Peggy's output would appear in her window at the same time as Vic's. The output would alternate from window to window as the messages are passed between Peggy and Vic. Peggy's output is as follows:

```
I will prove that 1339
        is a quadratic residue modulo 1411

------------------------------------------------------------
                Begin Round 1
------------------------------------------------------------
I set y = 400
Vic chose i = 1
I set z = 248
Vic's check SUCCEEDED!!
------------------------------------------------------------
                End Round 1
------------------------------------------------------------
------------------------------------------------------------
                Begin Round 2
------------------------------------------------------------
I set y = 562
Vic chose i = 1
I set z = 1096
Vic's check SUCCEEDED!!
------------------------------------------------------------
                End Round 2
------------------------------------------------------------

.
.
.

------------------------------------------------------------
                Begin Round 11
------------------------------------------------------------
I set y = 64
Vic chose i = 1
I set z = 1228
Vic's check SUCCEEDED!!
------------------------------------------------------------
                End Round 11
------------------------------------------------------------
```

The transcript of the protocol, saved in the file `quadres.trans` would appear as follows:

```
n = 1411
x = 1339
```

```
y1 = 400
i1 = 1
z1 = 248

y2 = 562
i2 = 1
z2 = 1096

⋮

y11 = 64
i11 = 1
z11 = 1228

SUCCESS
```

The proceeding was an example of Vic accepting Peggy's proof that x is a quadratic residue of n. If Vic rejects Peggy's proof, we get the following output from Vic:

```
Peggy needs a seed before she can decide what she will prove.
Number: 357
Which decision method is to be used? (1, 2, or 3): 2
Should n be the product of two primes? (y/n): y
Should n always contain the maximum number of digits? (y/n): n
Peggy sent n = 3683
           x = 1592
The transcript will be appended to the file 'quadres.trans'

⋮

------------------------------------------------------------
                 Begin Round 4
------------------------------------------------------------
Peggy sent y = 992
I set i = 1
Peggy sent z = 2220
My check FAILED:
        z^2 = 4928400
   IS NOT congruent to
        x^i*y = 1579264
                modulo 3683
------------------------------------------------------------
                 End Round 4
------------------------------------------------------------
1592 IS NOT a quadratic residue modulo 3683
```

The protocol continued for four rounds because Vic chose i to be zero in the first three rounds, but in the fourth round he chose i to be one. The only change in Peggy's output is that when one of the rounds fails, she prints `Vic's check FAILED!!` instead of `Vic's check SUCCEEDED!!`. The only change in the transcript is that it ends with the word `FAILURE` rather than `SUCCESS`. Note that the second input method was used to generate this example since the first input method always generates successful proofs.

The last sample that will be provided is more complex, n has two hundred digits rather than four digits. This size n would be appropriate for instances which may lead to applications. The last two rounds of Vic's output are presented here:

```
------------------------------------------------------------
                  Begin Round 661
------------------------------------------------------------
Peggy sent y = 29932935356102797305800321992560928991046145614 0
377526149689
I set i = 0
Peggy sent z = 54711000133522323780883380411 7
My check SUCCEEDED:
        z^2 = 29932935356102797305800321992560928991046145614 03
77526149689
                  IS congruent to
        x^i*y = 29932935356102797305800321992560928991046145614
0377526149689
                  modulo 13966970297595587619927559838946483564 53
525671942923533375787959536428912020962767278349966854337997743
277743677754117027228282947661993018419180213306116939036776165
589921682309023024916569765379552 9
------------------------------------------------------------
                  End Round 661
------------------------------------------------------------
------------------------------------------------------------
                  Begin Round 662
------------------------------------------------------------
Peggy sent y = 25659357872395319513656089247390328568187727879 8
520897175490966013932862454547973708066500984383795575605772903
366323009036931334477826524136915178707746868174405585802472477
713689
I set i = 0
Peggy sent z = 50655066747952587722741644869177520264114174406 5
899780652801322440536588652493494329856133
My check SUCCEEDED:
        z^2 = 25659357872395319513656089247390328568187727879859
208971754909660139328624545479737080665009843837955756057729033
663230090369313344778265241369151787077468681744055858024724777
```

```
13689
                IS congruent to
        x^i*y = 25659357872395319513656089247390328568187727879
8520897175490966013932862454547973708066500984383795575605772 90
336632300903693133447782652413691517870774686817440558580247247
7713689
                modulo 1396697029759558761992755983894648356453
525671942923533375787959536428912020962767278349966854337997743
277743677754117027228282947661993018419180213306116939036776165
58992168230902302491656976537955 29
---------------------------------------------------------------
                  End Round 662
---------------------------------------------------------------
165324073271220881199714298466285221924032319188934427222429278
306446369975208941935949102678514754688884433141915325946663017
621956 IS a quadratic residue modulo 1396697029759558761992755 9
838946483564535256719429235333757879595364289120209627672783499
668543379977432777436777541170272282829476619930184191802133061
1693903677616558992168230902302491656976537955 29
```

This example was generated using the first input method, and n is the product of two primes. The last two rounds of Peggy's input is as follows:

```
---------------------------------------------------------------
                  Begin Round 661
---------------------------------------------------------------
I set y = 29932935356102797305800321992560928991046145614037752
6149689
Vic chose i = 0
I set z = 547110001335223237808833804117
Vic's check SUCCEEDED!!
---------------------------------------------------------------
                  End Round 661
---------------------------------------------------------------
---------------------------------------------------------------
                  Begin Round 662
---------------------------------------------------------------
I set y = 256593578723953195136560892473903285681877278798520 89
71754909660139328624545479737080665009843837955756057729033663 2
300903693133447782652413691517870774686817440558580247247771368 9
Vic chose i = 0
I set z = 5065506674795258772274164486917752026411417440658997 8
06528013224405365886524934943298561 33
Vic's check SUCCEEDED!!
---------------------------------------------------------------
                  End Round 662
---------------------------------------------------------------
```

B.2.2 qrforge

The program qrforge implements the forging algorithm presented in Figure 3.2. The purpose of the forging algorithm is for Vic to be able to generate a transcript of the proof, that is identical to the real transcript, without any interaction with Peggy.

To demonstrate how qrforge works, here is a simple example. The following is the output that appears to the screen. The values of n and x are entered by the user. Note that these are the values of n and x that were used in the first example in Section B.2.1.

```
The inputs to this program are two integers n and x,
such that x is a quadratic residue mod n.
n: 1411
x: 1339

Need a seed before the transcript can be generated.
Number: 786
```

The following is the transcript which is printed to the file qrforge.trans:

```
n = 1411
x = 1339

y1 = 1024
i1 = 0
z1 = 134

y2 = 561
i2 = 1
z2 = 102

 ⋮

y11 = 559
i11 = 0
z11 = 95
```

B.3 Subgroup Membership

The programs subgrp and sgforge are very similar to quadres and qrforge in the way that they are implemented and in the way that they are run.

B.3.1 subgrp

The program subgrp implements the zero-knowledge protocol for subgroup membership in Figure 3.3. Like quadres, subgrp is a simple shell script which runs two programs subgrp_peggy and subgrp_vic.

Again, to illustrate how these programs work, a couple samples will be given. The first example is a simple example where Vic accepts Peggy's proof that b is a member of the subgroup of $\mathcal{Z}_n^*$ generated by a. The following is the output from Vic. Again the user must provide information when prompted by `subgrp_vic`.

```
Peggy needs a seed before she can decide what she will prove.
Number: 145
Which decision method is to be used? (1, 2, or 3): 2
Should n be the product of two primes? (y/n): y
Should n always contain the maximum number of digits?  (y/n): n
Peggy sent n = 77
           a = 47
           m = 30
           b = 9
The transcript will be appended to the file 'subgrp.trans'
------------------------------------------------------------
                 Begin Round 1
------------------------------------------------------------
Peggy sent g = 12
I set i = 0
Peggy sent h = 25
V: SUCCESS:
        a^h = 12
                IS congruent to
        b^i*g = 12
                modulo 77
------------------------------------------------------------
                 End Round 1
------------------------------------------------------------
------------------------------------------------------------
                 Begin Round 2
------------------------------------------------------------
Peggy sent g = 69
I set i = 1
Peggy sent h = 13
V: SUCCESS:
        a^h = 5
                IS congruent to
        b^i*g = 621
                modulo 77
------------------------------------------------------------
                 End Round 2
------------------------------------------------------------

:
```

```
------------------------------------------------------------
                Begin Round 7
------------------------------------------------------------
Peggy sent g = 71
I set i = 0
Peggy sent h = 18
V: SUCCESS:
        a^h = 71
                IS congruent to
        b^i*g = 71
                modulo 77
------------------------------------------------------------
                End Round 7
------------------------------------------------------------

9 IS a member of the subgroup generated by 47
```

Peggy's output is as follows:

```
I will prove that 9
        is a member of the subgroup of Zn* generated by 47
        where n = 77

------------------------------------------------------------
                End Round 1
------------------------------------------------------------
I set g = 12
Vic chose i = 0
I set h = 25
Vic's check SUCCEEDED!!
------------------------------------------------------------
                End Round 1
------------------------------------------------------------
------------------------------------------------------------
                End Round 2
------------------------------------------------------------
I set g = 69
Vic chose i = 1
I set h = 13
Vic's check SUCCEEDED!!
------------------------------------------------------------
                End Round 2
------------------------------------------------------------

 ⋮
```

```
------------------------------------------------------------
                End Round 7
------------------------------------------------------------
I set g = 71
Vic chose i = 0
I set h = 18
Vic's check SUCCEEDED!!
------------------------------------------------------------
                End Round 7
------------------------------------------------------------
```

The transcript from this protocol is saved in the file `subgrp.trans`. The transcript for this example is:

```
n = 77
a = 47
m = 30
b = 9

g1 = 12
i1 = 0
h1 = 25

g2 = 69
i2 = 1
h2 = 13

 ⋮

g7 = 71
i7 = 0
h7 = 18

SUCCESS
```

When using the first input method for `quadres`, the user is not required to enter any of the numbers, the program generates both n and x. The first input method for `subgrp` requires that the user enter n, a, and a number m such that $a^m \equiv 1 \bmod n$. It is recommended that n contain at least a few hundred digits in any instance that may lead to an application, and for large n it may be difficult to find m. The following sample discusses one method of generating examples with large n's

The next sample uses the first input method, so n, a, and m needed to be generated. This was done by constructing n to be the product of primes, p and q, and then randomly selecting a's until one was found such that

$$a^{\frac{(p-1)(q-1)}{2}} \equiv 1 \bmod n.$$

Since there is a high probability of this being true, it was not hard to find such an a. The last two rounds of Vic's output for this sample are as follows:

```
------------------------------------------------------------
                Begin Round 663
------------------------------------------------------------
Peggy sent g = 38154331188771523619612618196000479398684535296633440753282112920938836808571975087142291804580485603680319302297685882907801055771379579511624112231026371055574659348910499351689822731243480480810210
I set i = 0
Peggy sent h = 8
V: SUCCESS:
        a^h = 38154331188771523619612618196000479398684535296633440753282112920938836808571975087142291804580485603680319302297685882907801055771379579511624112231026371055574659348910499351689822731243480480810210
                IS congruent to
        b^i*g = 38154331188771523619612618196000479398684535296633440753282112920938836808571975087142291804580485603680319302297685882907801055771379579511624112231026371055574659348910499351689822731243480480810210
                modulo 433015474246692291195940810642831013612299092855267188898859535707794823648800476653153869928700599675355534589690384806843314627095767473084815123701682784453975925449236979291457254549488316969753310
------------------------------------------------------------
                End Round 663
------------------------------------------------------------
------------------------------------------------------------
                Begin Round 664
------------------------------------------------------------
Peggy sent g = 36824683715337210413095282596813173815190436652853123051951207376326877399959042487430624702497482743652378685211794121401530017583585726914221550021786577048145140823115355146495676960153783479835345
I set i = 0
Peggy sent h = 37425
V: SUCCESS:
        a^h = 36824683715337210413095282596813173815190436652853123051951207376326877399959042487430624702497482743652378685211794121401530017583585726914221550021786577048145140823115355146495676960153783479835345
                IS congruent to
        b^i*g = 36824683715337210413095282596813173815190436652853123051951207376326877399959042487430624702497482743652378685
```

```
211794121401530017583585726914221550021786577048145140823115355
146495676960153783479835345
                modulo 433015474246692291195940810642831013612 2
990928552671888988595357077948236488004766531538699287005996753
555345896903848068433146270957674730848151237016827844539759254
492369792914572545494883169697533 1
-----------------------------------------------------------------
                End Round 664
-----------------------------------------------------------------
333320461416341583382511951027959431743620882585142570095421964
380107446623664065092968608662338613142502027330572289091996894
932895260272285693577198550646671153015356670528910383386442413
38884378281 IS a member of the subgroup generated by 3484759052
927768818706895056053199717157684000970105451873189686944441635
890731700185740403989276101
```

The last two rounds of Peggy's output are:

```
-----------------------------------------------------------------
                End Round 663
-----------------------------------------------------------------
I set g = 381543311887715236196126181960004793986845352966334402
753282112920938836808571975087142291804580485603680319302297685
882907801055771379579511624112231026371055574659348910499351689
822731243480480810210
Vic chose i = 0
I set h = 8
Vic's check SUCCEEDED!!
-----------------------------------------------------------------
                End Round 663
-----------------------------------------------------------------
-----------------------------------------------------------------
                End Round 664
-----------------------------------------------------------------
I set g = 368246837153372104130952825968131738151904366528531230
519512073763268773999590424874306247024974827436523786852117944
121401530017583585726914221550021786577048145140823115355146495
676960153783479835345
Vic chose i = 0
I set h = 37425
Vic's check SUCCEEDED!!
-----------------------------------------------------------------
                End Round 664
-----------------------------------------------------------------
```

B.3.2 sgforge

The program sgforge is an implementation of the forging algorithm presented in Figure 3.4. The following is a sample of how the program sgforge works. The user must enter the values of n, a, b, and m. For this sample, the values from the first sample in Section B.3.1 were entered. The portion of the output that is printed to the screen is as follows:

```
The inputs to this program are four integers n, a, b, and m,
such that a^m = 1 mod n and gcd(b,n) = 1.
n: 77
a: 47
b: 9
m: 30

Need a seed before the transcript can be generated.
Number: 18
```

The transcript that is generated is stored in the file sgforge.trans. The transcript for this sample is:

```
n = 77
a = 47
b = 9
m = 30

g1 = 45
i1 = 1
h1 = 27

g2 = 67
i2 = 0
h2 = 20

 ⋮

g7 = 36
i7 = 1
h7 = 28
```

Appendix C

Source Code

C.1 Polynomial and Formula Header Files

The following are the header files for the classes Polynomial and Formula. These classes are used by the program qbf.

```
//
// Name:        poly.h
// Author:      Molli Noland
// Date:        May 2, 1999
// Description: A class that implements polynomials of a single variable.
//

#ifndef POLY_H
#define POLY_H

#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <rw/cstring.h>
#include "big.h"
#include "conversion.h"

#define MAX_POLY_LENGTH 30

class Polynomial {

  public: // constructors and destructors

    //
    // Name:        Constructor
    // Description: Initialize the polynomial.
    //
    Polynomial();
```

```
  //
  // Name:         Copy Constructor
  // Description: Copies the value of the argument to this polynomial.
  // Arguments:   The polynomial to copy.
  //
  Polynomial( const Polynomial &poly );

  //
  // Name:         Destructor
  // Description: Does nothing.
  //
  ~Polynomial();

public: // accessors

  //
  // Name:         get_value
  // Description: Gets the value of the polynomial when var_value is
  //              substituted for the variable.
  // Arguments:   The value to substitute.
  // Returns:     The value of the polynomial evaluated at var_value.
  //
  Big get_value( Big var_value );

public: // mutators

  //
  // Name:         set_variable
  // Description: Sets the name of the variable to var.
  // Arguments:   The name of the variable.
  //
  void set_variable( RWCString var );

  //
  // Name:         add
  // Description: Adds a single term to the polynomial.  If there is
  //              not a like term, and the polynomial is already of
  //              maximum length, nothing is added.
  // Arguments:   The coefficent and the power describing the term.
  //
  void add( Big coeff, Big power );

public: // operators

  //
  // Name:         Assignment Operator
  // Description: Assigns the value of the argument to this polynomial.
  // Arguments:   The polynomial to copy.
  // Returns:     A copy of this polynomial.
  //
```

```
    const Polynomial &operator=( const Polynomial &poly );

    //
    // Name:        Addition Operator
    // Description: Adds poly to this polynomial and returns the sum.
    // Arguments:   The polynomial to add to this.
    // Returns:     The sum polynomial.
    //
    Polynomial operator+( const Polynomial &poly );

    //
    // Name:        Multiplication Operator
    // Description: Multiplies this polynomial by poly and returns the
    //              product.
    // Arguments:   The polynomial to multiply by this.
    // Returns:     The product polynomial.
    //
    Polynomial operator*( const Polynomial &poly );

  public: // output

    //
    // Name:        Output Operator
    // Description: Outputs the polynomial to the output stream.
    //              Written in terms of the variable.
    // Arguments:   The output stream and the polynomial to output.
    // Returns:     The output stream.
    //
    friend ostream &operator<<( ostream &os,
                                const Polynomial &poly );

  private: // helper functions

    //
    // Name:        copy
    // Description: Called by the copy constructor and the assignment
    //              operator to copy the value of the argument to this
    //              polynomial.
    // Arguments:   The polynomial to copy.
    //
    void copy( const Polynomial &poly );

  private: // data members

    Big       the_poly[MAX_POLY_LENGTH];  // The coefficents and powers.
    int       length;                     // The length of the_poly.
    RWCString variable;                   // The name of the variable.

};
```

```
#endif


//
// Name:        formula.h
// Author:      Molli Noland
// Date:        May 2, 1999
// Description: A class that implements quantified Boolean formulas.
//

#ifndef FORMULA_H
#define FORMULA_H

#include <iostream.h>
#include <rw/cstring.h>
#include "big.h"
#include "poly.h"
#include "conversion.h"

#define MAX_FORMULA_LENGTH 100

class Formula {

  public: // constructors and destructors

    //
    // Name:        Constructor
    // Description: Sets attributes to their initial values.
    //
    Formula();

    //
    // Name:        Copy Constructor
    // Description: Copies the argument to this formula.
    //
    Formula( const Formula &formula );

    //
    // Name:        Destructor
    // Description: Does nothing.
    //
    ~Formula();

  public: // accessors

    //
    // Name:        get_value
    // Description: Returns the value of the formula.
    // Returns:     The value as a Big integer.
    //
```

```
Big get_value();

//
// Name:        get_length
// Description: Returns the length of the formula.
// Returns:     The length as an integer.
//
int get_length();

//
// Name:        poly_description
// Description: Returns the polynomial description of the formula.
//              Only recalculated if the formula has changed since
//              the last time it was called.
// Returns:     The polynomial object which describes the formula.
//
Polynomial poly_description();

//
// Name:        get_prime_version
// Description: Calculates the prime version of this formula.  The
//              prime version is found by removing the first
//              quantifier.
// Arguments:   The prime version is saved in a_prime, the variable
//              quantified by the removed quantifier is saved in
//              variable.  These are not set if there is no
//              quantifier to remove.
// Returns:     1 if the removed quantifier was universal,
//              0 if the removed quantifier was existential,
//              or -1 if there was no quantifier to remove.
//
int get_prime_version( Formula &a_prime, RWCString &variable );

//
// Name:        split_formula
// Description: Splits the formula at the first quantifier.
// Arguments:   Everything before the last & or | before the first
//              quantifier is saved in a1, and everything after the
//              & or | is saved in a2.
// Returns:     1  if the last operation before the first quantifier
//                 was &,
//              0  if the operation was |, or
//              -1 if there was no operation before the first
//                 quantifier.
//
int split_formula( Formula &a1, Formula &a2 );

//
// Name:        removed_parens
// Description: Indicates whether or not outside parens have been
```

```
  //              removed.
  // Returns:     Returns true if the outside parens were removed by
  //              the last call to remove_outside_parens (and the
  //              formula hasn't been changed since), and returns flase
  //              otherwise.
  //
  int removed_parens();

public: // mutators

  //
  // Name:        remove_outside_parens
  // Description: Removes the parens from the outside of the formula if
  //              they exist, and sets removed_parens to the
  //              appropriate value.
  //
  void remove_outside_parens();

  //
  // Name:        remove_quantifier
  // Description: Removes the first quantifier and replaces every
  //              occurrence of the variable that was quantified by the
  //              first quantifier with var_value.  It also replaces
  //              the negated occurrences of the variable with one
  //              minus var_value.
  // Arguments:   The value to replace the variable with, assumed to be
  //              zero or one.
  //
  void remove_quantifier( char var_value );

  //
  // Name:        append
  // Description: Appends the given string on the end of the formula if
  //              the formula is not already of maximum length.
  // Arguments:   The string to append to the formula.
  //
  void append( RWCString string );

  //
  // Name:        evaluate
  // Description: Replaces each occurrence of variable with var_value,
  //              and each occurrence of the negated variable with one
  //              minus var_value.
  // Arguments:   The variable to replace, and the value to replace it
  //              with.
  //
  void evaluate( RWCString variable, Big var_value );

public: // operators
```

```
  //
  // Name:         Assignment Operator
  // Description: Assigns the value of the argument to this formula.
  // Arguments:   The formula to copy.
  // Returns:     A reference to this formula.
  //
  const Formula &operator=( const Formula &formula );

public: // input and output

  //
  // Name:         Output Operator
  // Description: Outputs the formula to the output stream.
  // Arguments:   The oput stream and the formula to output.
  // Returns:     The output stream.
  //
  friend ostream &operator<<( ostream &os,
                              const Formula &formula );

  //
  // Name:         Input Operator.
  // Description: Reads in the formula from the input stream.
  //              The formula is assumed to have valid syntax.
  // Arguments:   The input stream and a reference to the
  //              formula object to read in.
  // Returns:     The input stream.
  //
  friend istream &operator>>( istream &is, Formula &formula );

private: // helper functions

  //
  // Name:         copy
  // Description: Called by the copy constructor and the assignment
  //              operator to copy the argument to this formula.
  // Arguments:   The formula to copy.
  //
  void copy( const Formula &formula );

  //
  // Name:         create_poly
  // Description: A recursive function called by poly_description which
  //              creates the polynomial associated with this formula.
  // Arguments:   The variable the polynomial is assumed to be in.
  // Returns:     The polynomial.
  //
  Polynomial create_poly( RWCString var );

private: // data members
```

```
    RWCString  the_formula[MAX_FORMULA_LENGTH]; // Formula as strings.
    int        length;             // Number of strings in the_formula.
    Big        value;              // Value of the formula.
    Polynomial poly;               // The associated polynomial.
    int        evaluated;          // True if value calculated.
    int        poly_created;       // True if polynomial created.
    int        parens_removed;     // True if outside parens removed.
};

#endif
```

C.2 qbf

The program qbf implements the protocol for determining the truth value of a QBF in Figure 2.3.

```
//
// Name:         qbf.cpp
// Author:       Molli Noland
// Date:         May 2, 1999
// Description:  Implementation of Shamir's Interactive Proof of
//               Quantified Boolean Formulas (QBF).
// Arguments:    The name of a file that contatins a QBF (this is
//               required), a prime (this is optional unless the value is
//               provided), and the proposed value of the QBF (this is
//               optional).
// Input:        The user will be asked for a seed to begin the random
//               number generation.
// Output:       The input to the protocol and the output at each round
//               of the protocol.  The style reflects the style of the
//               algorithm presented in this thesis.
//

#include "qbf.h"

int main( int argc, char *argv[] ){
  Miracl     precision(500,10);
  Big        p, a, r, orig_a, a1_inv;
  Formula    A, A1, A2, orig_A, A_prime;
  Polynomial q;
  int        done, accepts, operation;
  long       seed;
  fstream    input;
  RWCString  variable;

  // Check for the correct number of arguments.  If not correct,
  // print usage information and quit.
  if( ( argc < 2 ) || ( argc > 4 ) ){
    cout << "Usage:  qbf <filename> <prime> <value> where\n";
    cout << "    <filename> contains a Quantified Boolean ";
    cout << "Formula," << endl;
    cout << "    <prime> is a prime (optional unless value ";
    cout << "provided)," << endl;
    cout << "    and <value> is the value of the QBF, this is ";
    cout << "optional." << endl;
    return 0;
  }

  // Peggy needs to get a seed from Vic for choosing random numbers.
  cout << "Peggy needs a seed before she can choose ";
  cout << "random numbers." << endl << "Number: ";
```

```
cin >> seed;
irand( seed );

// Read in the QBF.
input.open( argv[1], ios::in );
input >> A;
input.close();

// Choose the prime if one is not provided.
if( argc < 3 )
  p = nextprime( rand( MAX_DIGITS, 10 ) );
else p = atoi( argv[2] );

// Set proposed value of A to 0, if a value is not provided.
if( argc < 4 ) a = 0;
else a = atoi( argv[3] );

// Store the initial versions of A and a.
orig_A - A;
orig_a = a;

// Output the input to the protocol.
cout << endl << "Input:" << endl;
cout << "-----" << endl;
cout << "   A: " << A << endl;
cout << "   p: " << p;

//
// This is the main loop of the protocol.
//
done   0;
accepts = 0;
while( !done ){

  //
  // Step A:  Peggy sends a (mod p) to Vic.  She splits A into
  //          A1 and A2 and sends that information to Vic.
  //

  // Output header.
  cout << endl << "Peggy:" << endl;
  cout << "-----" << endl;
  cout << "   a (mod p): " << ( a % p );

  // Split formula and print A1 and A2.
  operation = A.split_formula( A1, A2 );
  cout << "   A1: " << A1 << endl;
  cout << "   A2: " << A2 << endl;

  // Print out what A equals in terms of A1 and A2.
```

```
switch( operation ){
  case 1:
    cout << "   A = A1 * A2" << endl;
    break;
  case 0:
    cout << "   A = A1 + A2" << endl;
    break;
  default:
    if( A1.get_length() == 0 ) cout << "   A = A2" << endl;
    else cout << "   A = A1" << endl;
    break;
}

//
// Step B:  If A2 is empty, Vic checks if he is done.
//          Else if A1 is not empty Vic sets A = A2 and
//              computes the new value of a.
//          Else he gets more information from Peggy.
//

// Output header.
cout << endl << "Vic:" << endl;
cout << "---" << endl;

// If A2 is empty, Vic accepts if and oly if a = a1 mod p.
if( A2.get_length() == 0 ){
  done = 1;
  if( ( a % p ) == ( A1.get_value() % p ) ) accepts = 1;
  else accepts = 0;
}

// If A1 is not empty, set A to A2 and correct value of a.
else if( A1.get_length() != 0 ){
  A   A2;

  // If the operation was multiplication and a1 = 0,
  // then Vic accepts if and only if a = 0 mod p.
  if( ( operation == 1 ) && ( A1.get_value() == 0 ) ){
    done = 1;
    if( ( a % p ) == 0 ) accepts - 1;
    else accepts = 0;
  }

  // If the operation is multiplication, then a = a / a1 mod p.
  else if( operation == 1 ){
    a1_inv = inverse( A1.get_value(), p );
    a = ( a * a1_inv ) % p;
  }

  // If the operation is addition, then a = a - a1 mod p
```

```
  else if( operation == 0 ) a = ( a   A1.get_value() ) % p;

  // Print out the new values of A and a.
  cout << "   A: " << A << endl;
  cout << "   a (mod p): " << ( a % p );
}

// Otherwise, Vic needs to ask Peggy for more information.
else{
  cout << "   I need more information!!" << endl;

  //
  // Step 1:  Peggy sends polynomial description q of A'
  //          to Vic.
  //

  // Print header.
  cout << endl << "Peggy:" << endl;
  cout << "-----" << endl;

  // Peggy computes the polynomial description of A',
  // and prints out A' and its polynomial description.
  q = A.poly_description();
  operation = A.get_prime_version( A_prime, variable );
  cout << "   A': " << A_prime << endl;
  cout << "   q(" << variable << "): " << q << endl;

  //
  // Step 2:  Vic checks the value of a.  He then sends a random r,
  //          and resets the values of A and a depending on r.
  //

  // Print header.
  cout << endl << "Vic:" << endl;
  cout << "---" << endl;

  // If the removed quantifier was universal, Vic checks
  // that q(0) * q(1) = a mod p.  If not he rejects.
  if( ( operation == 1 ) &&
      ( ( ( q.get_value( 0 ) * q.get_value( 1 ) ) % p )
        != ( a % p ) ) ){
    done = 1;
    accepts = 0;
  }

  // If the removed quantifier was existential, Vic checks
  // that q(0) + q(1) = a mod p.  If not he rejects.
  else if ( ( operation == 0 ) &&
      ( ( ( q.get_value( 0 ) + q.get_value( 1 ) ) % p )
        != ( a % p ) ) ){
```

```
        done = 1;
        accepts = 0;
      }

      // If Vic has not rejected he chooses a random r and sets
      // A equal to A' evaluated at r, and sets a = q(r) mod p.
      else{
        r = rand(((brand() % (MAX_DIGITS / 2)) + 1), 10) % p;
        A_prime.evaluate( variable, r );
        A = A_prime;
        a = q.get_value( r ) % p;

        // Print out the new values of A and a.
        cout << "   A = A'(" << variable << " = ";
        cout << big_to_string( r ) << "): " << A << endl;
        cout << "   a: " << ( a % p );
      }
    }
  }

  // Output the final outcome of the proof.
  cout << endl << "   The value of " << orig_A;
  if( accepts ) cout << " IS ";
  else cout << " IS NOT ";
  cout << big_to_string( orig_a ) << " mod " << p << endl;

  return 0;
}
```

C.3 zero_knowledge.h

The file zero_knowledge.h is included by several programs. It contains the declarations for the functions which implement the various input methods for quadres and subgrp. The header file also contains the declarations of global constants.

```
//
// Name:         zero_knowledge.h
// Author:       Molli Noland
// Date:         May 2, 1999
// Description: Include file for all zero-knowledge programs.  Includes
//              the necessary include files, the macro definitions, and
//              the global variables.
//

#ifndef ZERO_H
#define ZERO_H

#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <math.h>
#include "big.h"

#define MAX_DIGITS    5
#define MAX_DIGITS_2  6
#define NUM_ENTRIES   50

//
// Name:         qr_method_one
// Description: Chooses n and u at random.  Calculates x = u^2 mod n.
//              Because of the way that the calculation is done, x will
//              always be a quadratic residue of n.  n is restricted to
//              MAX_DIGITS digits.
// Parameters:  References to n, x, and u, and optional integers
//              prime_product and always_max.  If prime_product is
//              TRUE, n will be the product of two primes.  The default
//              value is FALSE.  If always_max is TRUE, n will always be
//              the maximum number of digits. The default value if FALSE.
// Returns:     The values of n, x, and u, such that x = u^2 mod n.
//
void qr_method_one( Big &n, Big &x, Big &u,
                    int prime_product = 0, int always_max = 0 );

//
// Name:         qr_method_two
// Description: Chooses n and x at random.  Finds u, if it exists, such
//              that x = u^2 mod n.  Because the calculation of u is
//              slow, n is restricted to max_dig digits.  This function
```

```
//              will produce x's that sometimes are and sometimes aren't
//              quadratic residues mod n.
// Parameters:  References to n, x, and u, and optional integers
//              prime_product, always_max, and max_dig.  If prime_product
//              is TRUE, n will be the product of two primes.  The
//              default value is FALSE.  If always_max is TRUE, n will
//              always be the maximum number of digits.  The default
//              value if FALSE.
// Returns:     The values of n, x, and u, such that x = u^2 mod n.  If
//              x is not a quadratic residue, u will be set to a random
//              value.
//
void qr_method_two( Big &n, Big &x, Big &u, int prime_product = 0,
                    int always_max = 0, int max_dig = MAX_DIGITS_2 );

//
// Name:        qr_method_three
// Description: Reads n, x, and u from a file.  This file contains x's
//              that are both quadratic residues and non-residues of n.
//              There are NUM_ENTRIES entries in this file.
// Parameters:  References to n, x, and u, and optional integer
//              prime_product.  If prime_product is TRUE, n will be the
//              product of two primes.  The default value is FALSE.
// Returns:     The values of n, x, and u, such that x = u^2 mod n.  If x
//              is not a quadratic residue, u will be set to a random
//              value.
//
void qr_method_three( Big &n, Big &x, Big &u, int prime_product   0 );

//
// Name:        sg_method_one
// Description: Vic inputs n, a, and m, such that a^m = 1 mod n. Randomly
//              choose k, and calculate b, such that a^k = b mod n.
//              Because of the way that the calculation is done, b will
//              always be a member of the subgroup of Zn* generated by a.
//              n is restricted to MAX_DIGITS digits.
// Parameters:  References to n, a, m, b, and k.
// Returns:     The values of n, a, m, b, and k, such that a^m   1 mod n,
//              a^k = b mod n.
//
void sg_method_one( Big &n, Big &a, Big &m, Big &k, Big &b );

//
// Name:        sg_method_two
// Description: Chooses n, a, and b at random, such that gcd(a,n) =
//              gcd(b,n) = 1.  Finds m and k, if k exists, such that
//              a^m = 1 mod n and a^k = b mod n. Because the calculations
//              are slow, n is restricted to max_dig digits.  This
//              function will produce b's that sometimes are and
//              sometimes aren't subgroup members.
```

```
// Parameters:  References to n, a, m, b, and k, and optional integers
//              prime_product, always_max, and max_dig.  If prime_product
//              is TRUE, n will be the product of two primes. The default
//              value is FALSE.  If always_max is TRUE, n will always
//              have max_dig digits.  The default value is FALSE.
// Returns:     The values of n, a, m, b, and k, such that a^m = 1 mod n,
//              a^k = b mod n, and gcd(a,n) = gcd(b,n) = 1.  If
//              b is not a subgroup member, k will be set to a random
//              value.
//
void sg_method_two( Big &n, Big &a, Big &m, Big &k, Big &b,
                    int prime_product = 0, int always_max = 0,
                    int max_dig = MAX_DIGITS_2 );

//
// Name:        sg_method_three
// Description: Reads n, a, m, k, and b from a file.  This file contains
//              b's that are both subgroup members and non-members. There
//              are NUM_ENTRIES entries in the file.
// Parameters:  References to n, a, m, b, and k, and an optional integer
//              prime_product.  If prime_product is TRUE, n will be the
//              product of two primes.  The default value is FALSE.
// Returns:     The values of n, a, m, b, and k, such that a^m = 1 mod n,
//              a^k = b mod n, and gcd(a,n) = gcd(b,n) = 1.  b is not a
//              subgroup member, k will be set to a random value.
//
void sg_method_three( Big &n, Big &a, Big &m, Big &k, Big &b,
                      int prime_product = 0 );

#endif
```

C.4 quadres

The program quadres implements the zero-knowledge protocol for quadratic residues in Figure 3.1. quadres is a simple shell script which simultaneously runs the programs quadres_peggy and quadres_vic in separate X-terminals.

```
#
# Name:         quadres
# Author:       Molli Noland
# Date:         May 2, 1999
# Description:  This program runs the prover and verifier programs for
#               the perfect zero-knowledge proof for quadratic residues
#               simultaneously.  The programs run in separate x-terms,
#               demonstrating the interaction between Peggy and Vic.
#

# This line initializes the signal file, in case the program has been
# run before.
echo "Peggy's COMPLETELY done" > quadres.signal;

# Run Peggy and Vic's programs in separate x-terms.
xterm -geometry 78x50+0+210 -sb -e quadres_vic &
xterm -geometry 78x50+577+210 -sb -e quadres_peggy &
```

```
//
// Name:         quadres_peggy.cpp
// Author:       Molli Noland
// Date:         May 2, 1999
// Description: Program which implements the prover's side of a
//               zero-knowledge proof for quadratic residues.
// Output:       Peggy's side of the correspondence.  The program
//               always terminates in an infinite loop, so that the user
//               can review the contents of the window that it is run in.
//               To exit the program, close the window.
//

#include "zero_knowledge.h"

main(){
  Miracl   precision(800,10);
  Big      n, x, v, y, z, u;
  long     seed;
  int      i, j, status, log2n, numdig, done, system_status;
  int      always_max, method, prime_product;
  fstream  vic_message, signal;
  FILE     *peggy_message;

  //
  // Initialization:  Peggy declares what she is proving to Vic.
  //
```

```
// Peggy has to wait until Vic has written his answers, so that she
// can decide what she is going to prove.
done = 0;
while( !done ){
  system_status = system( "grep Vic quadres.signal > quadres.sigout");
  signal.open( "quadres.sigout", ios::in );
  if( signal.peek() == 'V' ) done = 1;
  signal.close();
}

// Once she receives the signal, she reads Vic's message.
vic_message.open( "quadres_vic.msg", ios::in );
if( !vic_message ){
  cout << "Vic's message file could not be opened for reading, ";
  cout << "quitting now!!" << endl;
  while(1);
}
vic_message >> seed;
vic_message >> method;
vic_message >> prime_product;
vic_message >> always_max;
vic_message.close();

// Peggy uses the information she received from Vic to decide what to
// prove.
irand( seed );
switch( method ){
  case 1:
    qr_method_one( n, x, u, prime_product, always_max );
    break;
  case 2:
    qr_method_two( n, x, u, prime_product, always_max );
    break;
  case 3:
    qr_method_three( n, x, u, prime_product );
    break;
  default:
    cout << "Invalid selection.  Default is Method 1.\n\n";
    qr_method_one( n, x, u, prime_product, always_max );
    break;
}

// Peggy outputs what she is proving.
cout << endl << "I will prove that " << x;
cout << "        is a quadratic residue modulo " << n << endl;

// Write n and x to a file for Vic to read.
if( !( peggy_message = fopen( "quadres_peggy.msg", "w" ) ) ){
  cout << "Peggy's message file could not be opened for writing, ";
```

```
  cout << "quitting now!!" << endl;
  while(1);
}
cotnum( n.getbig(), peggy_message );
cotnum( x.getbig(), peggy_message );
fclose( peggy_message );

// Once this is done, she informs Vic that she is finished.
signal.open( "quadres.signal", ios::out );
if( !signal ){
  cout << "Signal file could not be opened for writing, quitting ";
  cout << "now!!" << endl;
  while(1);
}
signal << "Peggy's done" << endl;
signal.close();

// Peggy has to wait until Vic has received this information.
done = 0;
while( !done ){
  system_status = system( "grep Vic quadres.signal > quadres.sigout" );
  signal.open( "quadres.sigout", ios::in );
  if( signal.peek() == 'V' ) done = 1;
  signal.close();
}

//
// Step 1:  Repeat the process log n times.
//
j = 1;
log2n = bits(n);
status = 1;
while( ( j <= log2n ) && ( status ) ){

  cout << "-----------------------------------------------------\n";
  cout << "             Begin Round " << j << endl;
  cout << "-----------------------------------------------------\n";

  //
  // Step 1A:  Peggy chooses v, an element of Zn, at random, and
  //           then computes y = v^2 mod n, and sends it to Vic.
  //

  // Peggy chooses v and computes y.
  numdig = ( brand() % MAX_DIGITS ) + 1;
  v = rand( numdig, 10 ) % n;
  y = pow( v, 2, n );
  cout << "I set y - " << y;

  // Write y to a file for Vic to read.
```

```
if( !( peggy_message = fopen( "quadres_peggy.msg", "w" ) ) ){
  cout << "Peggy's message file could not be opened for writing, ";
  cout << "quitting now!!" << endl;
  while(1);
}
cotnum( y.getbig(), peggy_message );
fclose( peggy_message );

// Once this is done, she informs Vic that she is finished.
signal.open( "quadres.signal", ios::out );
if( !signal ){
  cout << "Signal file could not be opened for writing, quitting ";
  cout << "now!!" << endl;
  while(1);
}
signal << "Peggy's done" << endl;
signal.close();

//
// Step 1C:  Peggy wait's for Vic to send i, then Peggy computes
//           z = u^i*v mod n, where u is a quadratic residue of
//           x mod n.  Peggy sends z to Vic.
//

// Peggy has to wait until Vic has written i, and then she reads it in.
done = 0;
while( !done ){
  system_status = system( "grep Vic quadres.signal > quadres.sigout" );
  signal.open( "quadres.sigout", ios::in );
  if( signal.peek() == 'V' ) done = 1;
  signal.close();
}

// Peggy has received the signal, so she now reads the value of i.
vic_message.open( "quadres_vic.msg", ios::in );
if( !vic_message ){
  cout << "Signal file could not be opened for reading, quitting ";
  cout << "now!!" << endl;
  while(1);
}
vic_message >> i;
vic_message.close();
cout << "Vic chose i = " << i << endl;

// Peggy calculates z.
z = ((u^i) * v) % n;
cout << "I set z = " << z;

// Write z to a file for Vic to read.
if( !( peggy_message = fopen( "quadres_peggy.msg", "w" ) ) ){
```

```
    cout << "Peggy's message file could not be opened for writing, ";
    cout << "quitting now!!" << endl;
    while(1);
  }
  cotnum( z.getbig(), peggy_message );
  fclose( peggy_message );

  // Once this is done, she informs Vic that she is finished.
  signal.open( "quadres.signal", ios::out );
  if( !signal ){
    cout << "Signal file could not be opened for writing, quitting ";
    cout << "now!!" << endl;
    while(1);
  }
  signal << "Peggy's done" << endl;
  signal.close();

  // Peggy needs the status of the Vic's check in Step D.
  // She waits until he has sent the message.
  done = 0;
  while( !done ){
    system_status = system( "grep Vic quadres.signal > quadres.sigout" );
    signal.open( "quadres.sigout", ios::in );
    if( signal.peek() == 'V' ) done = 1;
    signal.close();
  }

  // Once he sends the message she reads the status.
  vic_message.open( "quadres_vic.msg", ios::in );
  if( !vic_message ){
    cout << "Signal file could not be opened for reading, quitting ";
    cout << "now!!" << endl;
    while(1);
  }
  vic_message >> status;
  vic_message.close();

  // If status == 1, this is a success, otherwise it is a failure.
  if( status )
    cout << "Vic's check SUCCEEDED!!" << endl;
  else
    cout << "Vic's check FAILED!!" << endl;

  cout << "-------------------------------------------------------\n";
  cout << "              End Round " << j << endl;
  cout << "-------------------------------------------------------\n";
  j += 1;
}

// Once this is done, she informs Vic that she is completely finished.
```

```
  signal.open( "quadres.signal", ios::out );
  if( !signal ){
    cout << "Signal file could not be opened for writing, quitting ";
    cout << "now!!" << endl;
    while(1);
  }
  signal << "Peggy's COMPLETELY done" << endl;
  signal.close();

  // Terminate the program with an infinite loop for reasons described
  // above.
  while(1);
}

//
// Name:        quadres_vic.cpp
// Author:      Molli Noland
// Date:        May 2, 1999
// Description: Program which implements the verifier's side of a
//              zero-knowledge proof for quadratic residues.
// Input:       The user needs to enter a seed to start the random number
//              generation, a number from 1 to 3 indicating the input
//              method, whether or not n is required to be the product
//              of two primes, and whether or not n is required to have
//              the maximum number of digits.
// Output:      Vic's side of the correspondence.  The program always
//              terminates in an infinite loop, so that the user can
//              review the contents of the window that it is run in.
//              To exit the program, close the window.  The transcript
//              of the proof is appended to the file "quadres.trans".
//

#include "zero_knowledge.h"

main(){
  Miracl    precision(800,10);
  Big       n, x, y, z, temp1, temp2;
  long      seed;
  char      primes, range;
  int       i, j, status, log2n, system_status, done;
  int       print_transcript, method, prime_product;
  int       always_max = 0;
  fstream   vic_message, peggy_message, signal;
  FILE      *trans;

  //
  // Initialization:  Vic needs to answer a series of questions so that
  //                  Peggy can decide what to prove.
  //
```

```
// A seed is needed to start the randomization process.
cout << "Peggy needs a seed before she can decide what ";
cout << "she will prove.\nNumber: ";
cin >> seed;
irand(seed);

//
// The verifier needs to decide what method, one, two, or three, Peggy
// will use to choose what she is going to prove.  He also needs to
// decide if n needs to be the product of two primes or not. Descriptions
// of the methods are included with the functions that implement them.
//
cout << "Which decision method is to be used? (1, 2, or 3): ";
cin >> method;
cout << "Should n be the product of two primes? (y/n): ";
cin >> primes;
switch( primes ){
  case 'N':
  case 'n':
    prime_product = 0;
    break;
  case 'Y':
  case 'y':
    prime_product = 1;
    break;
  default:
    cout << "Invalid selection.  Default is 'No'.\n\n";
    prime_product = 0;
    break;
}

// As long as Peggy is not getting the input from a file, she needs to
// know if n should have the maximum number of digits.
if( method != 3 ){
  cout << "Should n always contain the maximum number of digits? (y/n): ";
  cin >> range;
  switch( range ){
    case 'N':
    case 'n':
      always_max = 0;
      break;
    case 'Y':
    case 'y':
      always_max = 1;
      break;
    default:
      cout << "Invalid selection.  Default is 'No'.\n\n";
      always_max = 0;
      break;
  }
```

```
}

// Vic writes his answers to a file for Peggy to read.
vic_message.open( "quadres_vic.msg", ios::out );
if( ! vic_message ){
  cout << "Vic's message file could not be opened for writing, ";
  cout << "quitting now!!" << endl;
  while(1);
}
vic_message << seed << endl;
vic_message << method << endl;
vic_message << prime_product << endl;
vic_message << always_max << endl;
vic_message.close();

// Once this is done, he informs Peggy that he is finished.
signal.open( "quadres.signal", ios::out );
if( ! signal ){
  cout << "Signal file could not be opened for writing, quitting ";
  cout << "now!!" << endl;
  while(1);
}
signal << "Vic's done" << endl;
signal.close();

// Vic has to wait until Peggy has written n and x, so that he will know
// what she is trying to prove.  Once she has done this, read in n and x.
done = 0;
while( !done ){
  system_status = system( "grep Peggy quadres.signal > quadres.sigout" );
  signal.open( "quadres.sigout", ios::in );
  if( signal.peek() == 'P' ) done = 1;
  signal.close();
}

// Once he receives the signal, he reads Peggy's message.
peggy_message.open( "quadres_peggy.msg", ios::in );
if( ! peggy_message ){
  cout << "Peggy's message file could not be opened for reading, ";
  cout << "quitting now!!" << endl;
  while(1);
}
peggy_message >> n;
peggy_message >> x;
peggy_message.close();
cout << "Peggy sent n = " << n;
cout << "           x = " << x;

// Open the file to print the transcript to and print the inputs to
// the proof.
```

```
if( trans = fopen( "quadres.trans", "a" ) ){
  cout << "The transcript will be appended to the file 'quadres.trans'\n";
  print_transcript = 1;
  fputs( "-------------------------------------------------\n", trans);
  fputs( "n = ", trans );
  cotnum( n.getbig(), trans );
  fputs( "x = ", trans );
  cotnum( x.getbig(), trans );
}
else{
  print_transcript = 0;
  cout << "Transcript file could not be opened.  ";
  cout << "No transcript will be printed. \n";
}

// Once this is done, he informs Peggy that he is finished.
signal.open( "quadres.signal", ios::out );
if( ! signal ){
  cout << "Signal file could not be opened for writing, quitting ";
  cout << "now!!" << endl;
  while(1);
}
signal << "Vic's done" << endl;
signal.close();

//
// Step 1:  Repeat the process log n times.
//
j = 1;
log2n = bits(n);
status = 1;
while( ( j <= log2n ) && ( status ) ){

  cout << "-------------------------------------------------------\n";
  cout << "              Begin Round " << j << endl;
  cout << "-------------------------------------------------------\n";

  //
  // Step 1B:  Vic receives y, and then Vic chooses i = 0 or 1 at
  //           random, and sends i to Peggy.
  //

  // Vic waits until Peggy has written y, and then reads it in.
  done = 0;
  while( !done ){
    system_status = system( "grep Peggy quadres.signal > quadres.sigout" );
    signal.open( "quadres.sigout", ios::in );
    if( signal.peek() == 'P' ) done = 1;
    signal.close();
  }
```

```
// Once he receives the signal, he can read Peggy's message.
peggy_message.open( "quadres_peggy.msg", ios::in );
if( ! peggy_message ){
  cout << "Peggy's message file could not be opened for reading, ";
  cout << "quitting now!!" << endl;
  while(1);
}
peggy_message >> y;
peggy_message.close();
cout << "Peggy sent y = " << y;

// Vic chooses i
i = brand() % 2;
cout << "I set i = " << i << endl;

// Vic writes i to a file for Peggy to read.
vic_message.open( "quadres_vic.msg", ios::out );
if( ! vic_message ){
  cout << "Vic's message file could not be opened for writing, ";
  cout << "quitting now!!" << endl;
  while(1);
}
vic_message << i << endl;
vic_message.close();

// Once this is done, he informs Peggy that he is finished.
signal.open( "quadres.signal", ios::out );
if( ! signal ){
  cout << "Signal file could not be opened for writing, quitting ";
  cout << "now!!" << endl;
  while(1);
}
signal << "Vic's done" << endl;
signal.close();

//
// Step 1D:  Vic receives z, and then Vic checks that
//           z^2 = x^i*y mod n. Returns true if the step is verified.
//

// Vic waits until Peggy has written z, and then reads it in.
done = 0;
while( !done ){
  system_status = system( "grep Peggy quadres.signal > quadres.sigout" );
  signal.open( "quadres.sigout", ios::in );
  if( signal.peek() == 'P' ) done = 1;
  signal.close();
}
```

```
// Once he receives the signal, he reads Peggy's message.
peggy_message.open( "quadres_peggy.msg", ios::in );
if( ! peggy_message ){
  cout << "Peggy's message file could not be opened for reading, ";
  cout << "quitting now!!" << endl;
  while(1);
}
peggy_message >> z;
peggy_message.close();
cout << "Peggy sent z = " << z;

// Vic performs the check.
temp1 = (z^2);
temp2 = (x^i) * y;

// Print a message indicating the outcome of the check.
if( ( temp1 % n ) != ( temp2 % n ) ){

  // Step is not verified -- print status message.
  status = 0;
  cout << "My check FAILED:" << endl;
  cout << "        z^2 = " << temp1;
  cout << "   IS NOT congruent to " << endl;
}
else{

  // Step is verified -- print status message.
  status = 1;
  cout << "My check SUCCEEDED:" << endl;
  cout << "        z^2 = " << temp1;
  cout << "                IS congruent to " << endl;
}
cout << "        x^i*y = " << temp2;
cout << "                modulo " << n;

cout << "-------------------------------------------------------\n";
cout << "              End Round " << j << endl;
cout << "-------------------------------------------------------\n";

// Print transcript of this round.
if( print_transcript ){
  fputs( "\n", trans );
  fprintf( trans, "y%d = ", j );
  cotnum( y.getbig(), trans );
  fprintf( trans, "i%d = %d\n", j, i );
  fprintf( trans, "z%d = ", j );
  cotnum( z.getbig(), trans );
}

// Vic writes the status of the check to a file for Peggy to read.
```

```
    vic_message.open( "quadres_vic.msg", ios::out );
    if( ! vic_message ){
      cout << "Vic's message file could not be opened for writing, ";
      cout << "quitting now!!" << endl;
      while(1);
    }
    vic_message << status << endl;
    vic_message.close();

    // Once this is done, he informs Peggy that he is finished.
    signal.open( "quadres.signal", ios::out );
    if( ! signal ){
      cout << "Signal file could not be opened for writing, quitting ";
      cout << "now!!" << endl;
      while(1);
    }
    signal << "Vic's done" << endl;
    signal.close();

    j += 1;
  }

  //
  // Step 2:  Vic accepts Peggy's proof if Step D is verified
  //          in every round.
  //

  // Print a message indicating the outcome.
  cout << x;
  if( status ){
    cout << "        IS a quadratic residue modulo ";
    if( print_transcript )
      fprintf( trans, "\nSUCCESS\n" );
  }
  else{
    cout << "        IS NOT a quadratic residue modulo ";
    if( print_transcript )
      fprintf( trans, "\nFAILURE\n" );
  }
  cout << n << endl;

  // Close transcript file.
  if( print_transcript ){
    fclose( trans );
  }

  // End the program with an infinte loop as described above.
  while(1);
}
```

C.5 qrforge

This program implements the forging algorithm for the zero-knowledge proof for quadratic residues in Figure 3.2.

```
//
// Name:        qrforge.cpp
// Author:      Molli Noland
// Date:        May 2, 1999
// Description: Program which implements a forging algorithm for the
//              zero-knowledge proof of quadratic residues.  The
//              input to the program is given by the verifier.
// Input:       The user must enter n and x, such that n and x are
//              relatively prime.
// Output:      Appends the transcript of the forging algorithm to the
//              file "qrforge.trans".
//

#include "zero_knowledge.h"

int main(){
  Miracl      precision(800,10);
  const Big   ONE(1);
  Big         n, x, y, z, xinv;
  long        seed;
  int         i, j, log2n, numdig;
  FILE        *trans;

  // Get the inputs to the program.
  cout << "The inputs to this program are two integers n and x,\n";
  cout << "such that x is a quadratic residue mod n.\n";
  cout << "n: ";
  cin >> n;
  cout << "x: ";
  cin >> x;
  cout << endl;

  // The gcd of x and n must be one, if it is not, there is not an
  // inverse.  The inverse is neccessary for the forging algorithm.
  // If the gcd is not one, quit now!!
  if( gcd(x,n) != ONE ){
    cout << "The gcd of x and n is not one; and therefore the inverse \n";
    cout << "can not be calculated, and the forging algorithm fails!!\n";
    return 0;
  }
  else{
    xinv = inverse(x,n);
  }

  // Need to get a seed from Vic to generate the transcript.
```

```
cout << "Need a seed before the transcript can be generated.\n";
cout << "Number: ";
cin >> seed;
irand(seed);

// Open the output file for the transcript.
if( !( trans = fopen( "qrforge.trans", "a" ) ) ){
  cout << "Cannot open the output file 'qrforge.trans'.  Exiting!!\n";
  return 0;
}

//
// Step 1: Output n and x to the transcript.
//
fputs( "-------------------------------------------------\n", trans);
fputs( "n = ", trans );
cotnum( n.getbig(), trans );
fputs( "x = ", trans );
cotnum( x.getbig(), trans );

//
// Step 2:  Repeat the generation of the triples log n times.
//
log2n = bits(n);
for( j = 1; j <= log2n; j++ ){

  //
  // Steps 2A and 2B:  Randomly generate i and z.
  //
  i = brand() % 2;
  numdig = ( brand() % MAX_DIGITS ) + 1;
  z = rand( numdig, 10 ) % n;

  //
  // Step 2C:  Calculate y = z^2*inv(x^i) mod n.
  //
  y = (z^2) % n;
  if( i == 1 ) y = ( xinv * y ) % n;

  //
  // Step 2D:  Output the triple to the transcript.
  //
  fputs( "\n", trans );
  fprintf( trans, "y%d = ", j );
  cotnum( y.getbig(), trans );
  fprintf( trans, "i%d = %d\n", j, i );
  fprintf( trans, "z%d = ", j );
  cotnum( z.getbig(), trans );
}
```

```
  fclose( trans );
  return 0;
}
```

C.6 subgrp

The program subgrp implements the zero-knowledge protocol for subgroup membership in Figure 3.3. Like quadres, subgrp is a simple shell script which simultaneously runs two programs subgrp_peggy and subgrp_vic.

```
#
# Name:         subgrp
# Author:       Molli Noland
# Date:         May 2, 1999
# Description:  This program runs the prover and verifier programs for
#               the perfect zero-knowledge proof for subgroup membership
#               simultaneously.  The programs run in separate x-terms,
#               demonstrating the interaction between Peggy and Vic.
#

# This line initializes the signal file, in case the program has been
# run before.
echo "Peggy's COMPLETELY done" > subgrp.signal;

# Run Peggy and Vic's programs in separate x-terms.
xterm -geometry 78x50+0+210 -sb -e subgrp_vic &
xterm -geometry 78x50+577+210 -sb -e subgrp_peggy &
```

```
//
// Name:         subgrp_peggy.cpp
// Author:       Molli Noland
// Date:         May 2, 1999
// Description: Program which implements the prover's side of a
//              zero-knowledge proof of subgroup membership.
// Output:       Peggy's side of the correspondence.  The program
//              always terminates in an infinite loop, so that the user
//              can review the contents of the window that it is run in.
//              To exit the program, close the window.
//

#include "zero_knowledge.h"

main(){
  Miracl    precision(800,10);
  Big       n, a, b, m, j, k, h, g;
  long      seed;
  int       i, l, status, log2n, numdig, always_max;
  int       method, prime_product, done, system_status;
  fstream   vic_message, signal;
  FILE      *peggy_message;

  //
  // Initialization:  Peggy declares what she is proving to Vic.
  //
```

```
// Peggy has to wait until Vic has written his answers, so that she
// can decide what she is going to prove.
done = 0;
while( !done ){
  system_status = system( "grep Vic subgrp.signal > subgrp.sigout");
  signal.open( "subgrp.sigout", ios::in );
  if( signal.peek() == 'V' ) done = 1;
  signal.close();
}

// Once Peggy receives the signal she reads Vic's message.
vic_message.open( "subgrp_vic.msg", ios::in );
if( !vic_message ){
  cout << "Signal file could not be opened for reading, quitting ";
  cout << "now!!" << endl;
  while(1);
}
vic_message >> seed;
vic_message >> method;
vic_message >> prime_product;
vic_message >> always_max;
vic_message.close();

// She then uses the information she received from Vic to decide what
// to prove.
irand( seed );
switch( method ){
  case 1:
    sg_method_one( n, a, m, k, b );
    break;
  case 2:
    sg_method_two( n, a, m, k, b, prime_product, always_max );
    break;
  case 3:
    sg_method_three( n, a, m, k, b, prime_product );
    break;
  default:
    cout << "Invalid selection.  Default is Method 1.\n\n";
    sg_method_one( n, a, m, k, b );
    break;
}

// Peggy outputs what she is proving.
cout << "I will prove that " << b;
cout << "        is a member of the subgroup of Zn* generated by " << a;
cout << "        where n = " << n << endl;

// Write n, a, b, and m to a file for Vic to read.
if( !( peggy_message   fopen( "subgrp_peggy.msg", "w" ) ) ){
```

```
  cout << "Peggy's message file could not be opened for writing, ";
  cout << "quitting now!!" << endl;
  while(1);
}
cotnum( n.getbig(), peggy_message );
cotnum( a.getbig(), peggy_message );
cotnum( m.getbig(), peggy_message );
cotnum( b.getbig(), peggy_message );
fclose( peggy_message );

// Once this is done, she informs Vic that she is finished.
signal.open( "subgrp.signal", ios::out );
if( !signal ){
  cout << "Signal file could not be opened for writing, quitting ";
  cout << "now!!" << endl;
  while(1);
}
signal << "Peggy's done" << endl;
signal.close();

// Peggy has to wait until Vic has received this information.
done = 0;
while( !done ){
  system_status = system( "grep Vic subgrp.signal > subgrp.sigout" );
  signal.open( "subgrp.sigout", ios::in );
  if( signal.peek() == 'V' ) done = 1;
  signal.close();
}

//
// Step 1:  Repeat the process log n times.
//
l = 1;
log2n = bits(n) ;
status  1;
while( ( l <= log2n ) && ( status ) ){

  cout << "-------------------------------------------------------\n";
  cout << "              End Round " << l << endl;
  cout << "-------------------------------------------------------\n";

  //
  // Step 1A:  Peggy chooses j, element of Zm, at random and computes
  //           g = a^j mod n, and sends g to Vic.
  //

  // Choose j and calculate g.
  numdig = ( brand() % MAX_DIGITS ) + 1;
  j = rand( numdig, 10 ) % m;
  g = pow(a, j, n);
```

```
cout << "I set g = " << g;

// Write g to a file for Vic to read.
if( !( peggy_message = fopen( "subgrp_peggy.msg", "w" ) ) ){
  cout << "Peggy's message file could not be opened for writing, ";
  cout << "quitting now!!" << endl;
  while(1);
}
cotnum( g.getbig(), peggy_message );
fclose( peggy_message );

// Once this is done, she informs Vic that she is finished.
signal.open( "subgrp.signal", ios::out );
if( !signal ){
  cout << "Signal file could not be opened for writing, quitting ";
  cout << "now!!" << endl;
  while(1);
}
signal << "Peggy's done" << endl;
signal.close();

//
// Step 1C:  Peggy waits for Vic to send i, then she computes
//           h = j + ik mod m, and sends h to Vic.
//

// Peggy has to wait until Vic has written i, and then she reads it in.
done = 0;
while( !done ){
  system_status = system( "grep Vic subgrp.signal > subgrp.sigout" );
  signal.open( "subgrp.sigout", ios::in );
  if( signal.peek() == 'V' ) done = 1;
  signal.close();
}

// Once Peggy receives the signal, she reads Vic's message.
vic_message.open( "subgrp_vic.msg", ios::in );
if( !vic_message ){
  cout << "Signal file could not be opened for reading, quitting ";
  cout << "now!!" << endl;
  while(1);
}
vic_message >> i;
vic_message.close();
cout << "Vic chose i = " << i << endl;

// Peggy calculates h.
h = ( j + i*k ) % m;
cout << "I set h = " << h;
```

```
  // Write h to a file for Vic to read.
  if( !( peggy_message = fopen( "subgrp_peggy.msg", "w" ) ) ){
    cout << "Peggy's message file could not be opened for writing, ";
    cout << "quitting now!!" << endl;
    while(1);
  }
  cotnum( h.getbig(), peggy_message );
  fclose( peggy_message );

  // Once this is done, she informs Vic that she is finished.
  signal.open( "subgrp.signal", ios::out );
  if( !signal ){
    cout << "Signal file could not be opened for writing, quitting ";
    cout << "now!!" << endl;
    while(1);
  }
  signal << "Peggy's done" << endl;
  signal.close();

  // Peggy needs the status of the check.  She has to wait until Vic
  // sends the message, then she reads the status of the check.
  done = 0;
  while( !done ){
    system_status = system( "grep Vic subgrp.signal > subgrp.sigout" );
    signal.open( "subgrp.sigout", ios::in );
    if( signal.peek() == 'V' ) done = 1;
    signal.close();
  }

  // Once Peggy receives the signal, she reads Vic's message.
  vic_message.open( "subgrp_vic.msg", ios::in );
  if( !vic_message ){
    cout << "Signal file could not be opened for reading, quitting ";
    cout << "now!!" << endl;
    while(1);
  }
  vic_message >> status;
  vic_message.close();

  // If status == 1, this is a success, otherwise it is a failure.
  if( status )
    cout << "Vic's check SUCCEEDED!!" << endl;
  else
    cout << "Vic's check FAILED!!" << endl;

  cout << "-------------------------------------------------------\n";
  cout << "               End Round " << l << endl;
  cout << "-------------------------------------------------------\n";
  l += 1;
}
```

```
  // Once this is done, she informs Vic that she is completely finished.
  signal.open( "quadres.signal", ios::out );
  if( !signal ){
    cout << "Signal file could not be opened for writing, quitting ";
    cout << "now!!" << endl;
    while(1);
  }
  signal << "Peggy's COMPLETELY done" << endl;
  signal.close();

  // The program ends with an infinite loop as explained above.
  while(1);
}

//
// Name:         subgrp_vic.cpp
// Author:       Molli Noland
// Date:         May 2, 1999
// Description: Program which implements the verifier's side of a
//               zero-knowledge proof for subgroup membership.
// Input:        The user needs to enter a seed to start the random number
//               generation, a number from 1 to 3 indicating the input
//               method, whether or not n is required to be the product
//               of two primes, and whether or not n is required to have
//               the maximum number of digits.
// Output:       Vic's side of the correspondence.  The program always
//               terminates in an infinite loop, so that the user can
//               review the contents of the window that it is run in.
//               To exit the program, close the window.  The transcript
//               of the proof is appended to the file "subgrp.trans"
//

#include "zero_knowledge.h"

main(){
  Miracl    precision(800,10);
  Big       n, a, b, m, h, g, temp1, temp2;
  long      seed;
  int       i, l, status, log2n, system_status, done;
  int       print_transcript, method, prime_product;
  int       always_max = 0;
  char      primes, range;
  fstream   vic_message, peggy_message, signal;
  FILE      *trans;

  //
  // Initialization:  Vic needs to answer a series of questions so that
  //                  Peggy can decide what to prove.
  //
```

```
// Peggy needs to get a seed from Vic to decide what she is proving.
cout << "Peggy needs a seed before she can decide what she ";
cout << "will prove.\nNumber: ";
cin >> seed;
irand(seed);

//
// Vic needs to decide what method, one, two, or three, Peggy
// will use to choose what she is going to prove.  He also needs to
// decide if n needs to be the product of two primes or not.
// Descriptions of the methods are included with the functions that
// implement them.
//
cout << "Which decision method is to be used? (1, 2, or 3): ";
cin >> method;
if( ( method == 2 ) || ( method == 3 ) ){
  cout << "Should n be the product of two primes? (y/n): ";
  cin >> primes;
  switch( primes ){
    case 'N':
    case 'n':
      prime_product = 0;
      break;
    case 'Y':
    case 'y':
      prime_product = 1;
      break;
    default:
      cout << "Invalid selection.  Default is 'No'.\n\n";
      prime_product = 0;
      break;
  }

  // If Peggy needs to generate n, she needs to know if n is required
  // to contain the maximum number of digits.
  if( method == 2 ){
    cout << "Should n always contain the maximum number of digits?";
    cout << "  (y/n): ";
    cin >> range;
    switch( range ){
      case 'N':
      case 'n':
        always_max = 0;
        break;
      case 'Y':
      case 'y':
        always_max = 1;
        break;
      default:
```

```
        cout << "Invalid selection.  Default is 'No'.\n\n";
        always_max = 0;
        break;
    }
  }
}

// Vic writes his answers to a file for Peggy to read.
vic_message.open( "subgrp_vic.msg", ios::out );
if( ! vic_message ){
  cout << "Vic's message file could not be opened for writing, ";
  cout << "quitting now!!" << endl;
  while(1);
}
vic_message << seed << endl;
vic_message << method << endl;
vic_message << prime_product << endl;
vic_message << always_max << endl;
vic_message.close();

// Once this is done, he informs Peggy that he is finished.
signal.open( "subgrp.signal", ios::out );
if( ! signal ){
  cout << "Signal file could not be opened for writing, quitting ";
  cout << "now!!" << endl;
  while(1);
}
signal << "Vic's done" << endl;
signal.close();

// Vic has to wait until Peggy has written n, a, m, and b, so that he
// will know what she is trying to prove.  Once she has done this,
// read in n, a, m, and b.
done = 0;
while( !done ){
  system_status   system( "grep Peggy subgrp.signal > subgrp.sigout" );
  signal.open( "subgrp.sigout", ios::in );
  if( signal.peek() == 'P' ) done = 1;
  signal.close();
}

// Once Vic receives the signal, he reads Peggy's message.
peggy_message.open( "subgrp_peggy.msg", ios::in );
if( ! peggy_message ){
  cout << "Peggy's message file could not be opened for reading, ";
  cout << "quitting now!!" << endl;
  while(1);
}
peggy_message >> n;
peggy_message >> a;
```

```
peggy_message >> m;
peggy_message >> b;
peggy_message.close();
cout << "Peggy sent n = " << n;
cout << "            a = " << a;
cout << "            m = " << m;
cout << "            b = " << b;

// Open the file to print the transcript to.
if( trans   fopen( "subgrp.trans", "a" ) ){
  cout << "The transcript will be appended to the file 'subgrp.trans'\n";
  print_transcript = 1;
  fputs( "-------------------------------------------------\n", trans);
  fputs( "n = ", trans );
  cotnum( n.getbig(), trans );
  fputs( "a = ", trans );
  cotnum( a.getbig(), trans );
  fputs( "m = ", trans );
  cotnum( m.getbig(), trans );
  fputs( "b   ", trans );
  cotnum( b.getbig(), trans );
}
else{
  print_transcript = 0;
  cout << "Transcript file could not be opened.  ";
  cout << "No transcript will be printed. \n";
}

// Once this is done, he informs Peggy that he is finished.
signal.open( "subgrp.signal", ios::out );
if( ! signal ){
  cout << "Signal file could not be opened for writing, quitting ";
  cout << "now!!!" << endl;
  while(1);
}
signal << "Vic's done" << endl;
signal.close();

//
// Step 1:  Repeat the process log n times.
//
l = 1;
log2n = bits(n) ;
status = 1;
while( ( l <= log2n ) && ( status ) ){

  cout << "-------------------------------------------------------\n";
  cout << "               Begin Round " << l << endl;
  cout << "-------------------------------------------------------\n";
```

```
//
// Step 1B:  Vic waits to receive g, and then he chooses i = 0 or 1
//           at random.  He sends i to Peggy.
//

// Vic waits until Peggy has written g, and then reads it in.
done = 0;
while( !done ){
  system_status = system( "grep Peggy subgrp.signal > subgrp.sigout" );
  signal.open( "subgrp.sigout", ios::in );
  if( signal.peek() == 'P' ) done = 1;
  signal.close();
}

// Once he receives the signal, Vi reads Peggy's message.
peggy_message.open( "subgrp_peggy.msg", ios::in );
if( ! peggy_message ){
  cout << "Peggy's message file could not be opened for reading, ";
  cout << "quitting now!!" << endl;
  while(1);
}
peggy_message >> g;
peggy_message.close();
cout << "Peggy sent g = " << g;

// Vic chooses i
i = brand() % 2;
cout << "I set i = " << i << endl;

// Vic writes i to a file for Peggy to read.
vic_message.open( "subgrp_vic.msg", ios::out );
if( ! vic_message ){
  cout << "Vic's message file could not be opened for writing, ";
  cout << "quitting now!!" << endl;
  while(1);
}
vic_message << i << endl;
vic_message.close();

// Once this is done, he informs Peggy that he is finished.
signal.open( "subgrp.signal", ios::out );
if( ! signal ){
  cout << "Signal file could not be opened for writing, quitting ";
  cout << "now!!" << endl;
  while(1);
}
signal << "Vic's done" << endl;
signal.close();

//
```

```
// Step 1D:  Vic waits to receive h, then he checks that
//           a^h = b^i*g mod n.
//

// Vic waits until Peggy has written h, and then reads it in.
done = 0;
while( !done ){
  system_status = system( "grep Peggy subgrp.signal > subgrp.sigout" );
  signal.open( "subgrp.sigout", ios::in );
  if( signal.peek() == 'P' ) done = 1;
  signal.close();
}

// Once Vic receives the signal, he reads Peggy's message.
peggy_message.open( "subgrp_peggy.msg", ios::in );
if( ! peggy_message ){
  cout << "Peggy's message file could not be opened for reading, ";
  cout << "quitting now!!" << endl;
  while(1);
}
peggy_message >> h;
peggy_message.close();
cout << "Peggy sent h = " << h;

// Vic performs the check.
temp1 = pow(a, h, n);
temp2 = (b^i) * g;

// Print out the outcome of the check.
if( temp1 != (temp2 % n) ){

  // Step is not verified -- print status message.
  status = 0;
  cout << "V: FAILED:" << endl;
  cout << "        a^h = " << temp1;
  cout << "                IS NOT congruent to\n";
}
else{

  // Step is verified -- print status message.
  status = 1;
  cout << "V: SUCCESS:" << endl;
  cout << "        a^h - " << temp1;
  cout << "                IS congruent to\n";
}
cout << "        b^i*g = " << temp2;
cout << "                modulo " << n;

cout << "-------------------------------------------------------\n";
cout << "              End Round " << l << endl;
```

```
  cout << "-------------------------------------------------------\n";

  // Print transcript of this round
  if( print_transcript ){
    fputs( "\n", trans );
    fprintf( trans, "g%d = ", l );
    cotnum( g.getbig(), trans );
    fprintf( trans, "i%d = %d\n", l, i );
    fprintf( trans, "h%d = ", l );
    cotnum( h.getbig(), trans );
  }

  // Vic writes the status of the check to a file for Peggy to read.
  vic_message.open( "subgrp_vic.msg", ios::out );
  if( ! vic_message ){
    cout << "Vic's message file could not be opened for writing, ";
    cout << "quitting now!!" << endl;
    while(1);
  }
  vic_message << status << endl;
  vic_message.close();

  // Once this is done, he informs Peggy that he is finished.
  signal.open( "subgrp.signal", ios::out );
  if( ! signal ){
    cout << "Signal file could not be opened for writing, quitting ";
    cout << "now!!" << endl;
    while(1);
  }
  signal << "Vic's done" << endl;
  signal.close();

  l += 1;
}

//
// Step 2:  Vic accepts Peggy's proof if Step D is verified
//          in every round.
//

// Print the outcome of the proof.
cout << endl << b;
if( status ){
  cout << "        IS a member of the subgroup generated by " << a;
  if( print_transcript )
    fprintf( trans, "\nSUCCESS\n" );
}
else{
  cout << "        IS NOT a member of the subgroup generated by " << a;
  if( print_transcript )
```

```
      fprintf( trans, "\nFAILURE\n" );
    }
    cout << endl;

    // Close transcript file.
    if( print_transcript ){
      fclose( trans );
    }

    // The program ends with an infinite loop as explained above.
    while(1);
}
```

C.7 sgforge

The program sgforge is an implementation of the forging algorithm for the perfect zero-knowledge proof of subgroup membership, presented in Figure 3.4.

```
//
// Name:         sgforge.cpp
// Author:       Molli Noland
// Description: Program which implements a forging algorithm for the
//               zero-knowledge proof of subgroup membership.
// Input:        The values of n, a, b, and m, such that a^m   1 mod n,
//               and b and n are relatively prime.
// Output:       The transcript from the forging algorithm is appended
//               to the file "sgforge.trans".
//

#include "zero_knowledge.h"

int main(){
  Miracl      precision(800,10);
  const Big   ONE(1);
  Big         n, a, b, m, h, g, binv;
  long        seed;
  int         i, j, log2n, numdig;
  FILE        *trans;

  // Get the inputs to the program.
  cout << "The inputs to this program are four integers n, a, b, and m,\n";
  cout << "such that a^m = 1 mod n and gcd(b,n) = 1.\n";
  cout << "n: ";
  cin >> n;
  cout << "a: ";
  cin >> a;
  cout << "b: ";
  cin >> b;
  cout << "m: ";
  cin >> m;
  cout << endl;

  // Peggy checks that a^m = 1 mod n; if not the input was invalid.
  // NOTE: There is an ERROR here when n is an odd multiple of five!!
  if( pow( a, m, n ) != ONE ){
    cout << "The input was invalid!!" << endl;
    return 0;
  }

  // The gcd of b and n must be one, if it is not, there is not an
  // inverse. The inverse is neccessary for the forging algorithm.  If
  // the gcd is not one, quit now!!
  if( gcd(b,n) != ONE ){
```

```
  cout << "The gcd of b and n is not one; and therefore the inverse \n";
  cout << "can not be calculated, and the forging algorithm fails!!\n";
  return 0;
}
else{
  binv = inverse(b,n);
}

// Need to get a seed from Vic to generate the transcript.
cout << "Need a seed before the transcript can be generated.\n";
cout << "Number: ";
cin >> seed;
irand(seed);

// Open the output file for the transcript.
if( !( trans = fopen( "sgforge.trans", "a" ) ) ){
  cout << "Cannot open the output file 'sgforge.trans'.  Exiting!!\n";
  return 0;
}

//
// Step 1: Output n, a, b, and m to the transcript.
//
fputs( "-------------------------------------------------\n", trans);
fputs( "n = ", trans );
cotnum( n.getbig(), trans );
fputs( "a = ", trans );
cotnum( a.getbig(), trans );
fputs( "b = ", trans );
cotnum( b.getbig(), trans );
fputs( "m = ", trans );
cotnum( m.getbig(), trans );

//
// Step 2:  Repeat the generation of the triples log n times.
//
log2n = bits(n);
for( j = 1; j <= log2n; j++ ){

  //
  // Steps 2A and 2B:  Randomly generate h and i.
  //
  numdig = ( brand() % MAX_DIGITS ) + 1;
  h = rand( numdig, 10 ) % m;
  i = brand() % 2;

  //
  // Step 2C:  Calculate g = a^h*inv(b^i) mod n.
  //
  g = pow(a, h, n);
```

```
    if( i == 1 ) g = ( binv * g ) % n;

    //
    // Step 2D:  Output the triple to the transcript.
    //
    fputs( "\n", trans );
    fprintf( trans, "g%d = ", j );
    cotnum( g.getbig(), trans );
    fprintf( trans, "i%d = %d\n", j, i );
    fprintf( trans, "h%d = ", j );
    cotnum( h.getbig(), trans );
  }

  fclose( trans );
  return 0;
}
```

C.8 genfile

The program genfile creates the files used by the third input method for both quadres and subgrp.

```
//
// Name:         genfile.cpp
// Author:       Molli Noland
// Date:         May 2, 1999
// Description:  Creates a file used by Method 3 of quadres or subgrp.
//               Options include the maximum number of digits in n and
//               if it is necessary for n to be a product of primes.
// Arguments:    The program takes three arguments:
//                   1.  An integer saying which files to generate.
//                       1 -- quadres, 2-- subgrp, or 3 -- both?
//                   2.  The maximum number of digits for n (1 to 100).
//                   3.  If n needs to be the product of two primes (y/n).
//                   4.  If n is always max # of digits (y/n).
//                   5.  Seed for generating random numbers.
//

#include "zero_knowledge.h"

int main( int argc, char* argv[] ){
  Miracl      precision(800,10);
  int         program, max_dig, prime_product, i, always_max;
  char        primes;
  Big         n, x, u, a, b, m, k;
  FILE        *file;

  // Check that the correct number of arguments were passed in.
  // If not, print the usage information.
  if( argc != 6 ){
    cout << "Usage:  'genfile <files><max_size><prime_poduct><always_max>";
    cout << " <seed>' where\n";
    cout << "   <files> is 1 -- quadres, 2 -- subgrp, or 3 -- both,\n";
    cout << "   <max_size> is the maximum number of digits in n, \n";
    cout << "   <prime_product> indicates if n needs to be the product";
    cout << " of two primes,\n";
    cout << "   <always_max> indicates if n is always max # of digits, and\n";
    cout << "   <seed> is positive integer for random number generation.\n";
    return 0;
  }

  // Find out if we are creating a file for quadres for subgrp, or for
  // both.
  program = atoi( argv[1] );
  if( ( program < 1 ) || ( program > 3 ) ){
    program = 3;
    cout << "Invlaid file specifier.  Setting to default of 3." << endl;
```

```
}

// Find out the maximum number of digits in n, between 1 and 100.
max_dig = atoi( argv[2] );
if( ( max_dig > 100 ) || ( max_dig < 1 ) ){
  max_dig = MAX_DIGITS_2;
  cout << "Invalid # of digits.  ";
  cout << "Setting to default of " << max_dig << ".\n";
}

// Determine which file to create based on if n is required to be the
// product of two primes.
if( ( ( argv[3][0] == 'N' ) || ( argv[3][0] == 'n' ) ) &&
    ( argv[3][1] == '\0' ) ){
  prime_product = 0;
}
else if ( ( ( argv[3][0] == 'Y' ) || ( argv[3][0] == 'y' ) ) &&
          ( argv[3][1] == '\0' ) ){
  prime_product = 1;
}
else{
  cout << "Invalid answer to the question, if n must be the product\n";
  cout << "of two primes.  Setting to default of no." << endl;
  prime_product = 0;
}

// Determine if n is to always be the maximum number of digits or not.
if( ( ( argv[4][0] == 'N' ) || ( argv[4][0] == 'n' ) ) &&
    ( argv[4][1] == '\0' ) ){
  always_max = 0;
}
else if ( ( ( argv[4][0] == 'Y' ) || ( argv[4][0] == 'y' ) ) &&
          ( argv[4][1] == '\0' ) ){
  always_max = 1;
}
else{
  cout << "Invalid answer to the question, if n must be the maximum\n";
  cout << "number of digits.  Setting to default of no." << endl;
  always_max = 0;
}

// If n must be the product of primes, then max_digits must be greater
// than 1.
if( ( prime_product ) && ( max_dig == 1 ) ){
  cout << "n cannot be the product of two primes and be restricted to\n";
  cout << "   one digit, the maximum is being set to two.\n";
  max_dig = 2;
}

// Start the randomization process.
```

```
irand( atol( argv[5] ) );

// Create the quadres file if we are suppose to.
if( ( program == 1 ) || ( program == 3 ) ){

  // Open the file.
  if( prime_product ){
    if( !( file = fopen( "quadres.primes", "w" ) ) ){
      cout << "Cannot open the file!!  Exiting now!!" << endl;
      return 0;
    }
  }
  else{
    if( !( file = fopen( "quadres.input", "w" ) ) ){
      cout << "Cannot open the file!!  Exiting now!!" << endl;
      return 0;
    }
  }

  // Create the quadres file.
  for( i = 0; i < NUM_ENTRIES; i++ ){
    qr_method_two( n, x, u, prime_product, always_max, max_dig );

    // Output n, x, and u to the file.
    cotnum( n.getbig(), file );
    cotnum( x.getbig(), file );
    cotnum( u.getbig(), file );
    fputs( "\n", file );
  }

  fclose( file );
}

// Create the subgrp file if we are suppose to.
if( ( program == 2 ) || ( program == 3 ) ){

  // Open the file.
  if( prime_product ){
    if( !( file = fopen( "subgrp.primes", "w" ) ) ){
      cout << "Cannot open the file!!  Exiting now!!" << endl;
      return 0;
    }
  }
  else{
    if( !( file = fopen( "subgrp.input", "w" ) ) ){
      cout << "Cannot open the file!!  Exiting now!!" << endl;
      return 0;
    }
  }
```

```
    // Create the subgrp file.
    for( i = 0; i < NUM_ENTRIES; i++ ){
      sg_method_two( n, a, m, k, b, prime_product, always_max, max_dig );

      // Output n, a, m, k, and b to the file.
      cotnum( n.getbig(), file );
      cotnum( a.getbig(), file );
      cotnum( m.getbig(), file );
      cotnum( k.getbig(), file );
      cotnum( b.getbig(), file );
      fputs( "\n", file );
    }

    fclose( file );
  }

  return 0;
}
```

Bibliography

[AQIP99] *Second Workshop on Algorithms in Quantum Information Processing* (January 18-22, 1999, Chicago, IL). DePaul University School of Computer Science, Telecommunications, and Information Systems, http://www.cs.depaul.edu/AQIP99, 1999.

[Babai] Babai, L. "Trading group theory for randomness." In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing* (Providence, RI, May 6-8). New York: ACM, 1985, 421-429.

[Bennett] Bennett, C. H. and G. Brassard. "Quantum Cryptography: Public key distribution and coin tossing." In *Proceedings of the IEEE International Conference on Computers, Systems, and Signal Processing.* New York: IEEE, 1984, 175-179.

[Brassard90] Brassard, G. and C. Crépeau. "Sorting out zero-knowledge." *Lecture Notes in Computer Science* 434(1990): 181-191. (Advances in Cryptology - EUROCRYPT'89.)

[Brassard88] Brassard, G., D. Chaum and C. Crépeau. "Minimum disclosure proofs of knowledge." *Journal of Computer and Systems Science* 37(1988): 156-189.

[Crépeau] Crépeau, Claude. "A zero-knowledge Poker protocol that achieves confidentiality of the players' strategy or How to achieve an electronic Poker face." *Lecture Notes in Computer Science* 263(1987): 239-247. (Advances in Cryptology - CRYPTO'86.)

[Feige] Feige, U., A. Fiat and A. Shamir. "Zero-Knowledge Proofs of Identity." In *Proceedings 19th Annual Symposium on Theory of Computing.* New York: ACM, 1987, 210-217.

[Garey] Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W. H. Freeman and Company, 1979.

[Gilbert] Gilbert, John R., Thomas Lengauer and Robert Endre Tarjan. "The Pebbling Problem is Complete in Polynomial Space." *SIAM Journal of Computing* 9 (1980): 513-524.

[Goldreich] Goldreich, O., S. Micali and A. Wigderson. "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems." *Journal of the ACM* 38(1991): 691-729.

[Goldwasser85] Goldwasser, S., S. Micali and C. Rackoff. "The knowledge complexity of interactive proof systems." In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing* (Providence, RI, May 6-8). New York: ACM, 1985, 291-304.

[Goldwasser89] Goldwasser, S., S. Micali and C. Rackoff. "The knowledge complexity of interactive proof systems." *SIAM Journal of Computing* 18(1989):186-208.

[Goldwasser92] Goldwasser, S. and M. Sipser. "Private coins versus public coins in interactive proof systems." *Journal of the ACM* 39(1992): 859-868.

[Johnson] Johnson, David S. "The NP-Completeness Column: An Ongoing Guide." *Journal of Algorithms* 9(1988): 426-444.

[Lund] Lund, C., L. Fortnow, H. Karloff and N. Nisan. "Algebraic methods for interactive proof systems." In *Proceedings of 31st Symposium on Foundations of Computer Science.* New York: IEEE, 1989, 514-519.

[Menezes] Menezes, Alfred J., Paul C. van Oorschot and Scott A. Vanstone. *Handbook of Applied Cryptography.* New York: CRC Press, 1997.

[MIRACL] FTP site for MIRACL library. Dublin City University School of Computer Applications, ftp://ftp.compapp.dcu.ie/pub/crypto/miracl.zip.

[Montgomery] Montgomery, Peter. "Modular Multiplication Without Trial Division." *Mathematics of Computation* 44 (1985): 519-521.

[Moulton] Moulton, Norman W. "Algorithms for Multiple Precision Integer Math." Master's Project, R.I.T., 1998.

[Quisquater] Quisquater, J.-J. and L. Guillou. "How to explain zero-knowledge protocols to your children." *Lecture Notes in Computer Science* 435(1990): 628-631. (Advances in Cryptology - CRYPTO'89.)

[Rintanen] Rintanen, Jussi. *Evaluating QBF.* Universität Ulm Fakultät für Informatik, http://www.informatik.uni-ulm.de/ki/Rintanen/qbf.html, 1998.

[Salomaa] Salomaa, Arto. *Public-Key Cryptography.* New York: Springer, 1996.

[Schneier] Schneier, Bruce. *Applied Crytography: Protocols, Algorithms, and Source Code in C.* New York: John Wiley & Sons, Inc., 1996.

[Seberry] Seberry, Jennifer and Josef Pieprzyk. *Cryptography: An Introduction to Computer Security.* New York: Prentice-Hall, 1989.

[Shamir] Shamir, Adi. "IP = PSPACE." *Journal of the ACM* 39 (1992): 869-877.

[Shen] Shen, A. "IP = PSPACE: Simplified Proof." *Journal of the ACM* 39 (1992): 878-880.

[Shor] Shor, P. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer." *SIAM Journal of Computing* 26(1997): 1484-1509.

[Simmons] Simmons, Gustavus J. *Contemporary Cryptology: The Science of Information Integrity.* New York: IEEE Press, 1991.

[Sipser] Sipser, Michael. *Introduction to the Theory of Computation.* Albany: PWS Publishing Co., 1997.

[Stinson] Stinson, Douglas R. *Cryptography: Theory and Practice.* New York: CRC Press, 1995.

[Watrous] Watrous, John. "PSPACE has 2-round quantum interactive proof systems." Draft, Université de Montréal, 1999.

[Williams] Williams, Colin P. and Scott H. Clearwater. *Explorations in Quantum Computing.* New York: Springer-Verlag, 1998.